THE Multicore ASSOCIATION™

Multicore Communications API Specification V1.063 (MCAPI)

www.multicore-association.org

Multicore Communications API Specification V1.063

Document ID: MCAPI API Specification V1
Document Version: 1.063
Status: Release
Distribution: General

Name	Role
MCAPI WG Chair, Sven Brehmer	State of doc assessment
MCAPI WG members	Content, coverage and grammar
President of the Multicore Association, Markus Levy	Posting and Publication

Page Intentionally Left Blank

Table of Contents

1 History

Although multiprocessor designs have been around for decades, only recently have semiconductor vendors turned to multicore processors as a solution for the so-called Moore's Gap[1] effect. Simply put, modern multicore CPU design enables CPU performance to track Moore's law with attractive CPU power consumption properties. This trend is causing many system designers that previously would use single core designs to design multicore processors into hardware in order to meet their performance and/or power consumption requirements. This has a huge impact on the software architecture for those systems, which must now consider inter-processor communication (IPC) issues to pass data between the cores.

The Multicore Association's Communication API (MCAPI) traces its heritage to communications APIs such as MPI (Message Passing Interface) and Sockets. Both MPI and sockets were developed primarily with inter-computer communication in mind, while MCAPI is targeted primarily towards inter-core communication in a multicore chip. Accordingly, a principal design goal of MCAPI was to serve as a low-latency interface leveraging efficient on-chip interconnect in a multicore chip. MCAPI is thus a light-weight API whose communications latencies and memory footprint are expected to be significantly lower than that of MPI or Sockets. However, because of the more limited scope of multicore communications and its goal of low latency, MCAPI is less flexible than MPI or Sockets.

MCAPI can also be distinguished from POSIX (portable Unix) threads, pthreads. MCAPI is a communications API for messaging and streaming, while pthreads is an API for parallelizing or threading applications. Pthreads offers a programming model that relies on shared memory for communication and locks for synchronization. In multicore processors with caches, efficient pthreads implementations require support for cache coherence. Because they lack modularity, lock-based parallel programs are hard to compose together. MCAPI, with parallelism specified using standard means such as processes or threads, offers an alternative parallel programming approach that is modular and composable. Because it works both in multicore processors with only private memory or with shared memory, MCAPI allows simpler embedded multicore implementations.

IPC for device software utilizing multicore processors have some new requirements beyond those imposed on multiprocessor designs that utilized discrete processors. Within a multicore chip, the shorter distance for electrical signals enables data to be passed much more quickly, energy-efficiently, and reliably than between discrete processors on a board or over a backplane. In addition, the bandwidth available on chip is orders of magnitude greater. The low latency and high bandwidth offer the potential of ASIC-like (application specific integrated circuit) high-speed communication capabilities between cores. This means that the focus of IPC for multicore processors must prioritize performance in terms of both latency and throughput. Also, some multicore processors will have many cores that rely on chip-internal memory or cache to execute code (to avoid the von Neumann bottleneck which multicore processors can suffer due to the multiple cores contending for external memory accesses). Such cores that execute from chip-internal memory will require an IPC implementation that has a small memory footprint. For these reasons, the two primary goals for a multicore IPC API design are that the implementation of it can achieve extremely high performance and low memory footprint.

In order for a multicore IPC implementation to achieve high performance and tiny footprint, these two goals need to be immutable when in conflict with other desirable IPC API attributes such as physical or logical connectivity topology flexibility, ease of use, OS integration or compatibility with existing IPC programming models or APIs.

The Multicore Association's Communication API (MCAPI) endeavors to meet these goals. Before MCAPI, no IPC API has been designed with these two immutable goals targeting multicore processors.

[1] Moore's Law has hit a wall where increased transistors per sequential CPU chip is no longer resulting in a linear increase in performance of that CPU chip.

The MCAPI API was designed to be sufficiently complete for many multicore application programming tasks and also to serve as a solid foundation for sophisticated multicore software services. Therefore MCAPI can be used to implement distributed services such as name services and distributed work queues, as well as one sided services such as DMA.

2 Introduction

This document is intended to assist software developers who are either implementing MCAPI or writing applications that use MCAPI.

The MCAPI specification is both an API and communications semantic specification. It does not define which link management, device model or wire protocol is used underneath it. As such, by defining a standard API, it is intended to provide source code compatibility for application code to be ported from one operating environment to another.

MCAPI defines three fundamental communications types. These are:

1. Messages – connection-less datagrams.
2. Packet channels – connection-oriented, uni-directional, FIFO packet streams.
3. Scalar channels – connection-oriented single word uni-directional, FIFO packet streams.

Each of these communications types have their own API calls and are desirable for certain types of systems. Messages are the most flexible form of communication in MCAPI, and are useful when senders and receivers are dynamically changing and communicate infrequently. These are commonly used for synchronization and initialization.

Packet and scalar channels provide light-weight socket-like stream communication mechanisms for senders and receivers with static communication graphs. In a multicore, MCAPI's channel APIs provide an extremely low-overhead ASIC-like uni-directional FIFO communications capability. Channels are commonly set up once during initialization, during which the MCAPI runtime system attempts to perform as much of the work involved in communications (such as name lookup, route determination, and buffer allocation) between a specific pair of endpoints as possible. Subsequent channel sends and receives thus incur just the minimal overhead of physically transferring the data. Packet channels support streaming communication of multiword data buffers, while scalar channels are optimized for sequences of scalar values. Channel API calls are simple and statically typed, thereby minimizing dynamic software overhead, which allows applications to access the underlying multicore hardware with extremely low latency and energy cost.

MCAPI's objective is to provide a limited number of calls with sufficient communication functionality while keeping it simple enough to allow efficient implementations. Additional functionality can be layered on top of the API set. The calls are exemplifying functionality and are not mapped to any particular existing implementation at this stage.

3 MCAPI Overview

The major MCAPI concepts are covered in the following sections.

3.1 MCAPI Nodes

An MCAPI node is a logical abstraction that can be mapped to many entities, including but not limited to: a process, a thread, an instance of an operating system, a hardware accelerator, or a processor core. The `mcapi_initialize()` call takes a node number argument, and an MCAPI application may only call `mcapi_initialize()` once per node. It is an error to call `mcapi_initialize()` multiple times from a given thread of control. A given MCAPI implementation will specify what is a node (i.e., what thread of control – process, thread, or other -- is a node) for that implementation. A thread and process are just two examples of threads of control, and there could be others.

The MCAPI standard aims to be implementable on both process-oriented and thread-oriented systems. MCAPI defines a transport layer between 'nodes', which could be processes, threads, or some other thread-of-control abstraction. The standard explicitly avoids having any dependence on a particular shared memory model or process protection model. The communication operations defined by MCAPI should be implementable with identical semantics on either thread- or process- based systems. Thus, programs that use only the MCAPI APIs should be portable between MCAPI implementations. On the other hand, programs that make use of APIs outside of MCAPI, for example pthreads, will only be portable between systems that include those extra APIs.

THINGS TO REMEMBER:

- The MCAPI node-numbering plan is established when the MCAPI communication topology is configured at design-time, so programmers don't have to worry about this at run-time.
- It is up to the MCAPI implementation to configure communication topology interfaces to enable communication with other nodes in the communication topology.

3.2 MCAPI Endpoints

MCAPI endpoints are socket-like communication termination points. An MCAPI node can have multiple endpoints. Thus, an MCAPI endpoint is a topology-global unique identifier. In general, although users have flexibility in how they can create endpoints, MCAPI supports a common static way of naming an endpoint using a tuple of `<node_id, port_id>`. Static names allow the programmer to define a network topology at compile time and to embed the topology in the source code in a straightforward fashion. It also enables the use of external tools to generate topologies automatically and it facilitates simple initialization. Endpoints can also be created by requesting the next available endpoint which allows for topology flexibility.

Endpoints are identified by a `<node_id, port_id>` tuple. An endpoint is created by implicitly referring to the node_id of the node on which the endpoint is being created, and explicitly specifying a port_id. Alternatively, MCAPI allows the creation of an endpoint, by requesting the next available endpoint. The endpoint reference can be passed to other parties in the logical topology to dynamically change the logical communications topology. It is up to the implementation to ensure that the resulting endpoint reference is unique in the system. MCAPI allows a reference to endpoints to be obtained by a node other than the node whose node_id is associated with the endpoint and this endpoint reference can be used as a destination for messages or to specify the opposite end of a connection. Endpoint references created by requesting the next available endpoint must be passed by the creating node to other nodes to facilitate communication. Only the creating node is allowed to receive from an endpoint.

Endpoints may have a set of attributes. These attributes may be related to Quality of Service (QoS), buffers, timeouts, etc. Currently MCAPI defines a basic set of attribute numbers and their associated structure or data type for endpoints, found in the mcapi.h file, and implementations are free to add additional attribute numbers and data types as needed. MCAPI does define API functions to get and set the attributes of endpoints. Furthermore, it is an error to attempt a connection between endpoints

whose attributes are set in an incompatible way (for now, whether attributes are compatible or not is implementation defined). It is also an error to attempt to change the attributes of endpoints that are connected.

3.3 MCAPI Channels

Channels provide point-to-point FIFO connections between a pair of endpoints. MCAPI channels are unidirectional. There are two types of channels: packet channels and scalar channels. There is no separate channel object. Instead, channels can be referenced through the endpoints to which they are bound. Since channels are point-to-point connections between a pair of endpoints, either endpoint can be used to refer to the channel.

3.4 MCAPI Communications

Applications in a logical MCAPI topology communicate with one another by sending data between communication endpoints. Three modes of communication are supported: connectionless messages, connection-oriented packet channels, or connection-oriented scalar channels. Each communication action in all three methods (for example, a message send, a packet channel send, or a scalar channel send) requires the specification of a pair of endpoints – a send endpoint and a receive endpoint, explicitly for messages and implicitly for channels using a handle. The MCAPI runtime system implements each of these logical communication actions over some physical medium that is supported by MCAPI (such as on-chip interconnect, bus, shared memory, or Ethernet) by automatically establishing a “link” to enable communication with the other node in the network, and takes care of routing traffic over the appropriate link.

Messages and packet channels communicate using data buffers, while scalar channels transfer scalar values such as four-byte words.

From an application's perspective, a data buffer to be communicated is a byte string from 0 up to some maximum length determined by the amount of memory in the system, whose internal structure is determined by the application. This specification does not define a communication byte order because MCAPI is an API specification. Since MCAPI does not currently define a wire protocol (there may be a future Multicore Association specification which does specify that) then endian-ness and alignment issues etc are not relevant to this specification.

MCAPI communication takes place between endpoints. An endpoint is an entity that can send and receive data in either a connectionless or a connection-oriented manner.

Connectionless communication allows an endpoint to send or receive messages with one or more endpoints elsewhere in the communication topology. A given message can be sent to a single endpoint (unicast). Multicast and broadcast messaging modes are not supported by MCAPI, but can be built on top of it.

Connection-oriented communication allows an endpoint to establish a socket-like streaming connection to a peer endpoint elsewhere in the communication topology, and then send data to that peer. A connection is established using an explicit handshake mechanism prior to sending or receiving any application data. Once a connection has been established, it remains active for highly efficient data transfers until it is terminated by one of the sides, or until the communication path between the endpoints is severed (for example, by the failure of the node or link which one of the endpoints is utilizing).
Connection-oriented communication over MCAPI channels is designed to be reliable, in that an application can send data over a connection and assume that the data will be delivered to the specified destination as long as that destination is reachable.

THINGS TO REMEMBER:
- In an MCAPI communication topology where different nodes may be running on different CPU types and/or operating systems, applications must ensure that the internal structure of a message is well-defined, and account for any differences in message content endian-ness, field size and field alignment.

3.5 Data Delivery

On the surface, whether the communications use messages, packet channels or scalar channels, the data delivery in MCAPI is a simple series of steps: a sender creates and sends the data (either a buffer for messages and packet channels, or a scalar value for scalar channels), MCAPI carries the data to the specified destination, and the receiver receives and then consumes the data. In practice, this is exactly what happens most of the time. However, there are a number of places along the way where things can get complicated, and in these cases it is important for application designers to understand exactly what MCAPI will do.

The sections that follow describe the various steps performed by MCAPI during the transfer of a data packet.

3.5.1 Data Sending

The first step in sending data is to create it. The user application then sends the data using one of several mechanisms. The application supplies the data in a user buffer for messages and packet channels. The data is supplied directly as a scalar variable for scalar channels.

The most common reason MCAPI is unable to send a piece of data is because the sender passes in one or more invalid arguments to the send routine. The term "invalid" refers both to values that are never acceptable under any circumstances (such as specifying a data buffer size exceeding the maximum size allowed for a specific implementation) and to values that are not acceptable for the current sender (such as an invalid endpoint handle, or an unconnected channel).

In all of these cases the send operation will return a failure code indicating that the intended data was not sent. If the data is sent successfully, the send operation returns a success indication. Success means that the entire buffer has been sent.

THINGS TO REMEMBER:
- If the sender specifies a destination address, for a connectionless message, that does not currently exist within the MCAPI communication topology, MCAPI does *NOT* treat this as an invalid send request (i.e. it's not the sender's fault that the destination doesn't exist). Instead MCAPI creates the message and then sends it. The return value for the send operation will indicate success since the message was successfully created and processed by MCAPI. MCAPI cannot return an error since the connectivity failure could be occurring elsewhere in the MCAPI communication topology.

3.5.2 Data Routing

Once a send of some data has been requested, MCAPI then determines what node the data should be sent to. The endpoint address indicates the destination node to which the packet should be sent. The data is then either handed off to the destination endpoint directly if it is on the same node as the sender, or passed to a link for off-node transmission to the destination node.

One problem that can arise during the routing phase of packet delivery is that no working link to the specified destination node can be found. In this case, the packet is discarded.

3.5.3 Data Consumption

When the data reaches the destination endpoint, it is added to an endpoint receive queue. The data typically remains in the endpoint's receive queue until it is received by the application that owns the endpoint. Queued data items are consumed by the application in a FIFO manner. For messages, the

user application must supply a buffer into which the MCAPI runtime system fills in the data. For packet channels, on the other hand, the MCAPI runtime system supplies the buffer containing the data to the user. For messages, the application can use its data buffer after it is filled in by MCAPI in any way it chooses. Specifically, the application can re-supply the data buffer to MCAPI to receive the next message. Data buffers supplied by MCAPI during packet channel receives can also be used by the application in any way it chooses. MCAPI reuses a data buffer it has supplied only after the user application has specifically freed the buffer using the `mcapi_pktchan_free()` function.

If an application terminates access to the endpoint (using the `mcapi_pktchan_recv_close_i()`, `mcapi_pktchan_send_close_i()`, `mcapi_sclchan_recv_close_i()`, `mcapi_sclchan_recv_close_i()` API functions) before all data in the receive queue are consumed, all unconsumed data items are considered undeliverable and are discarded.

It is very important that MCAPI applications be engineered to consume their incoming data items at a rate that prevents them from accumulating in large numbers in any endpoint receive queue.

For the packet and scalar channels interfaces, a FIFO model is used. FIFOs have limited storage, and when the storage is used up, an implementation can chose to block until more storage is available or return an error code.
Non-blocking send operations will still return immediately, but data transfer will not occur if the FIFO is full. The sender will block upon calling `mcapi_wait()`, and will continue to block until a receive operation is performed or a timeout occurs.

The user can use the `mcapi_get/set_endpoint_attribute()` APIs to query or to control the amount of receive buffering that is provided for a given packet or scalar channel. Implementations may define a static receive buffer size according to the amount of buffering provided in drivers and/or hardware.

For asynchronous messages, on the other hand, an implementation could be allowed to dynamically allocate storage until the system runs out of memory. In this case, the message send function will return the MCAPI_ENO_MEM status code.

3.5.4 Waiting for non-blocking operations

The connectionless message and packet channel API functions have both blocking and non-blocking variants. The non-blocking variants of these MCAPI functions have "_i" appended to the function name to indicate that the function will return *immediately* and will complete in a non-blocking manner.

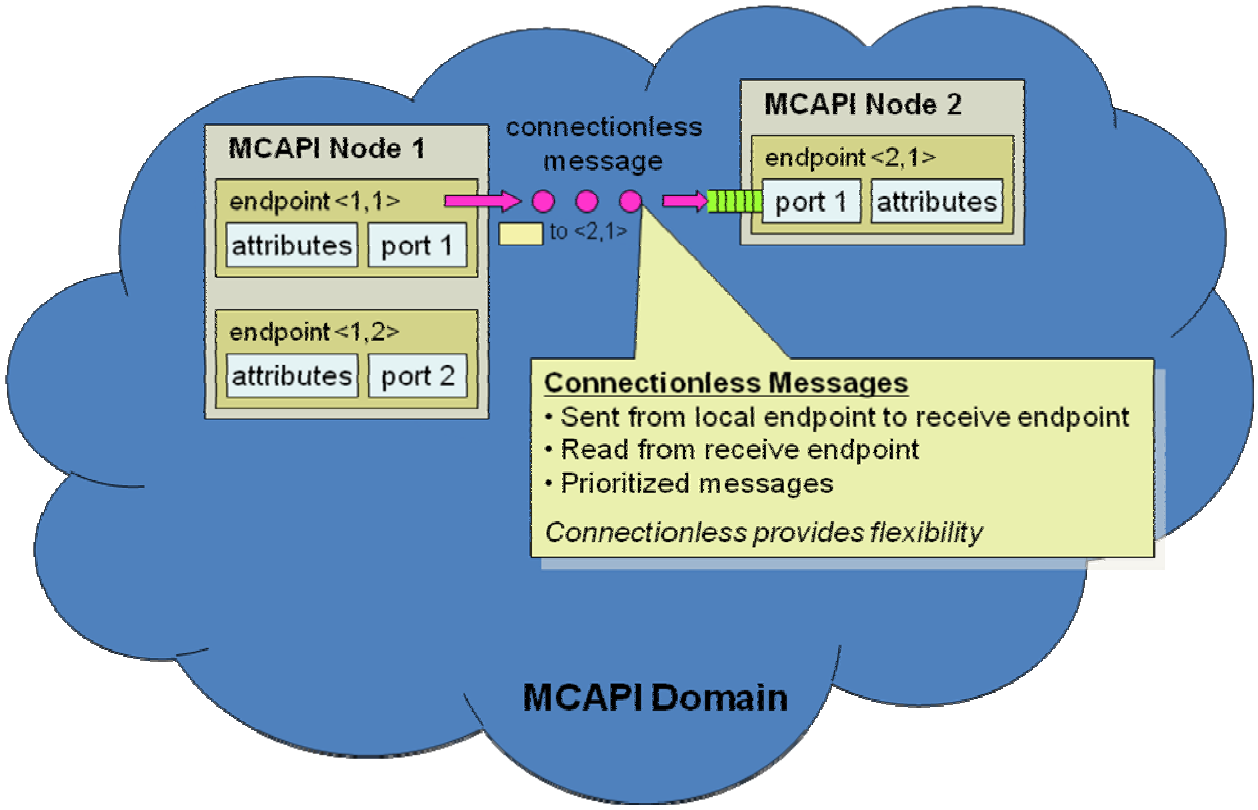
The non-blocking versions fill in a `mcapi_request_t` object and return control to the user before the communication operation is completed. The user can then use the `mcapi_test()`, `mcapi_wait()`, and `mcapi_wait_any()` functions to query the status of the non-blocking operation. These functions are non-destructive, meaning that no packet is removed from the endpoint queue by the functions. The `mcapi_test()` function is non-blocking whereas the `mcapi_wait()` and `mcapi_wait_any()` functions will block until the requested operation completes or a timeout occurs.

If a buffer of data is passed to a non-blocking operation (for example, to `mcapi_msg_send_i()` `mcapi_msg_recv_i()`, or to `mcapi_pktchan_send_i()`), that buffer may not be accessed by the user application for the duration of the non-blocking operation. That is, once a buffer has been passed to a non-blocking operation, the program may not read or write the buffer until `mcapi_test()`, `mcapi_wait()`, or `mcapi_wait_first()` have indicated completion, or until `mcapi_cancel()` has canceled the operation.

The MCAPI scalar channels API provides only blocking send and receive methods. Scalar channels are intended to provide a very low overhead interface for moving a stream of values. Non blocking operations add overhead. The sort of streaming algorithms that take advantage of scalar channels should not require a non-blocking send or receive method; each process should simply receive a value to work on, do its work, send the result out on a channel, and repeat. Applications that require non-blocking semantics should use packet channels instead of scalar channels.

3.6 MCAPI Messages

MCAPI messages provide a flexible method to transmit data between endpoints without first establishing a connection. The buffers on both sender and receiver sides must be provided by the user application. MCAPI messages may be sent with different priorities.

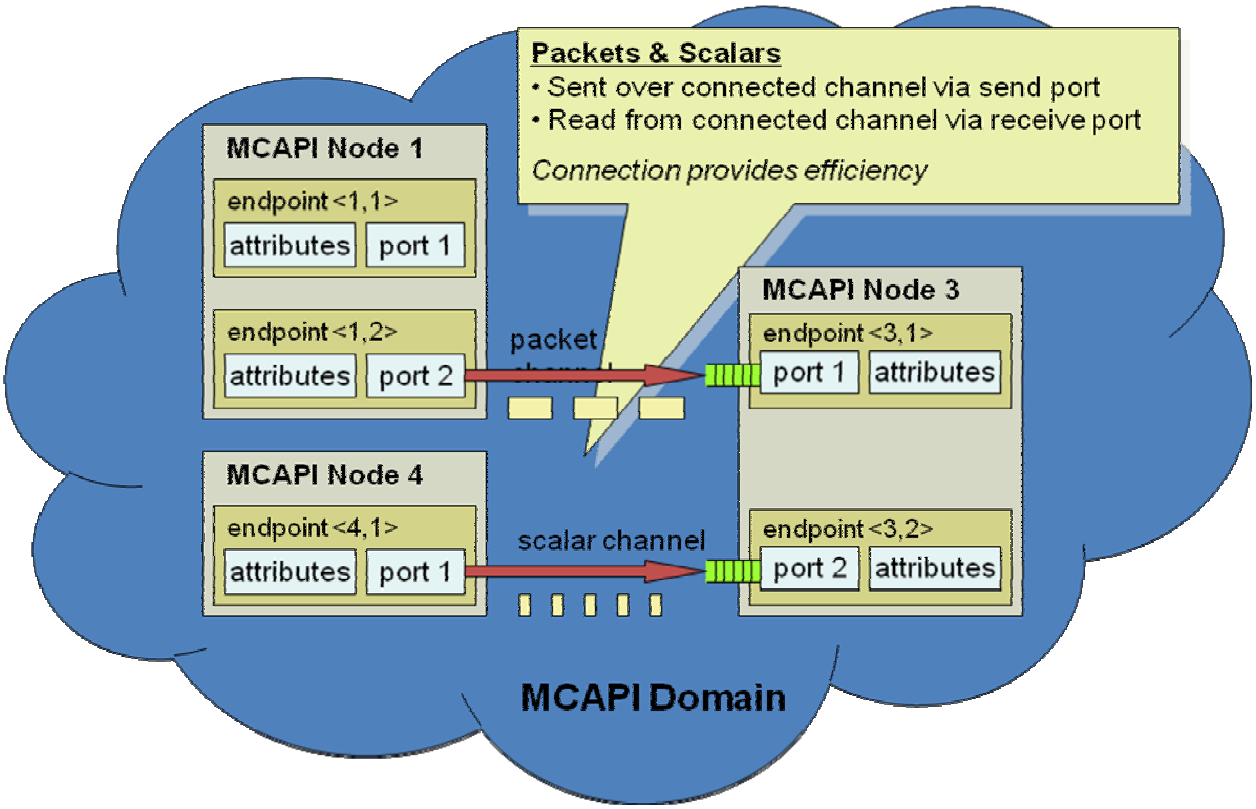


3.7 MCAPI Packet Channels

MCAPI packet channels provide a method to transmit data between endpoints by first establishing a connection, thus potentially removing the message header and route discovery overhead. Packet channels are unidirectional and deliver data in a FIFO (first in first out) manner. The buffers are provided by the MCAPI implementation on the receive side, and by the user application on the send side.

3.8 MCAPI Scalar Channels

MCAPI scalar channels provide a method to transmit scalars very efficiently between endpoints by first establishing a connection. Like packet channels, scalar channels are unidirectional and deliver data in a FIFO (first in first out) manner. The scalar functions come in 8-bit, 16-bit, 32-bit and 64-bit variants. The scalar receives must be of the same size as the scalar sends. A mismatch in size results in an error.



3.9 MCAPI Zero Copy

Zero copy means that data is passed by reference instead of a physical copy. This method is useful when multiple cores have shared memory, as one core can operate on a buffer of data and simply pass a pointer to the buffer to the next core, and save the time and energy of moving the data physically. Zero copy requires specific buffer management and is intended to be orthogonal to other API operations, to avoid complexity (simplifying code migration) and performance penalties for non-zero copy operations. The MCAPI message and packet channel mechanisms enable zero-copy systems by providing a transport layer for buffer metadata. It is expected that the underlying system provides the calls to allocate shared memory (to facilitate zero copy layering on top of MCAPI), aligned buffers, locks etc. MCAPI is a communications API and does not address these mechanisms. Accordingly, a zero copy messaging system can be layered on top of MCAPI by writing wrapper functions around the MCAPI send and receive functions.

3.10 MCAPI Error Handling Philosophy

Error handling is a fundamental part of the specification, however, some accommodations have been made to allow for trading off completeness for efficiency of implementation. For example, a couple of API functions allow implementations to optionally handle errors[2]. Consistent and efficient coding styles also governed the design of the error handling. In general, function calls include an error code parameter used by the API function to indicate detailed status. In addition, the return value of several API functions indicate success or failure which enables efficient coding practice. A value of type, `mcapi_status_t`, indicates success or failure states of API calls. `MCAPI_NULL` is a valid return value for `mcapi_status_t` (can be used for implementation optimization).

[2] Specifically, the `MCAPI_SCLCHAN_SEND_*` and `MCAPI_SCLCHAN_RECEIVE_*` routines define error arguments, but implementation are free to assume the routines always succeed.

If a process or thread attached to a node were to fail it is generally up to the application to recover from this failure. MCAPI provides timeouts for the `mcapi_wait()` and `mcapi_wait_any()` functions and provides the `mcapi_cancel()` function to clear outstanding non-blocking requests (at the non-failing side of the communication). It is also possible to reinitialize a failed node, by first calling `mcapi_finalize()`.

3.11 MCAPI Buffer Management Philosophy

MCAPI provides two APIs for transferring buffers of data between endpoints: MCAPI messages and MCAPI packet channels. Both APIs allow the programmer to send a buffer of data on one node and receive it on another node. The APIs differ in that messaging is connectionless, allowing any node to communicate with any other node at any time. Packet channels, on the other hand, perform repeated communication via a connection between two endpoints.

The APIs for sending a buffer via messaging or packet channels are very similar. Both APIs have a send method that allows the programmer to specify a buffer of data with two parameters: (void*, size_t). The programmer may send any data buffer they choose. There is no requirement that the buffer be allocated by MCAPI or returned to MCAPI after the send call completes.

The messaging and packet channels APIs differ in their handling of received buffers. The messaging API provides a "user-specified buffer" communications interface – the programmer specifies an empty buffer to be filled with incoming data on the receive side. The programmer can specify any buffer they choose and there is no requirement to allocate or release the buffer via an MCAPI call.

On the other hand, packet channels provide a "system-specified buffer" interface – the receive method returns a buffer of data at an address chosen by the runtime system. Since the receive buffer for a packet channel is allocated by the system, the programmer must return the buffer to the system by calling `mcapi_pktchan_free()`.

MCAPI data buffers may have arbitrary alignment. However, MCAPI does define two macros to help provide buffer alignment in performance-critical cases. Use of these macros is optional and can be defined with no value ("no-op"); implementers are required to handle send and receive buffers of arbitrary alignment, but can optimize for aligned buffers.

MCAPI implementations define the following two macros:

1. `MCAPI_DECL_ALIGNED` : a macro placed on static declarations in order to force the compiler to make the declared object aligned. For example:

```
mcapi_int_t MCAPI_DECL_ALIGNED my_int_to_send; /* See mcapi.h */
```

2. `MCAPI_BUF_ALIGN` : a macro that evaluates to the number of bytes to which dynamically allocated buffers should be aligned. For example:

```
memalign(MCAPI_BUF_ALIGN, sizeof(my_message));
```

MCAPI does not provide for any maximum buffer size. Applications may use any message size they choose, but implementations may fail with `MCAPI_ENO_MEM` if the system runs out of allocatable memory. Thus, the maximum size of a message in any particular implementation is limited by the amount of system memory available to that implementation.

3.12 Timeout/Cancellation Philosophy

The MCAPI API provides timeout functionality for its non-blocking calls through the timeout parameters of the `mcapi_wait()` and `mcapi_wait_any()` functions. For blocking functions implementations can optionally provide timeout capability through the use of endpoint attributes.

3.13 Backpressure mechanism

In many systems, the sender and receiver must coordinate to ensure that the messages are handled without overwhelming the receiver. Without such coordination, the sender must implement elaborate recovery mechanisms in case a message is dropped by the receiver. The overhead to handle these error cases is higher than pausing for some time and continuing later. The sender handles such scenarios by inquiring the receiver status before sending a message. The receiver returns the status based on the number of available buffers or buffer size.

The sender throttles the messages using the `mcapi_get_endpoint_attribute`() API and the `MCAPI_ATTR_RECV_BUFFERS_AVAILABLE attribute`. If required, the sender and receiver can reserve some buffers for higher priority messages. Further, a more sophisticated mechanism can be build on top of this API. For example, zones (green, yellow and red) can be defined using the number of available messages. The sender can use the zones to throttle the messages.

3.14 MCAPI Implementation Concerns

3.14.1 Link Management

The process for link configuration and link management are not specified within the MCAPI API spec. However, it is important to note that MCAPI does not define APIs or services (such as membership services) related to dynamic link state detection, node or service discovery and node or service state management. This is because MCAPI is designed to address multicore and multiprocessor systems where the number of nodes and their connectivity topology is known when the MCAPI system is configured. Since the MCAPI API specification does not specify any APIs that deal with network interfaces, an MCAPI implementation is free to manage links underneath the interface as it sees fit. In particular, an MCAPI implementation is able to restrict the topologies supported or the number of links between nodes in an MCAPI communication topology.

3.14.2 Thread Safe Implementations

MCAPI implementations are assumed to be reentrant (thread safe). Essentially, if an MCAPI implementation is available in a threaded environment, then it needs to be thread safe. MCAPI implementations can also be available in non-threaded environments, but the provider of such implementations will need to clearly indicate that the implementation is not thread safe.

3.15 MCAPI Potential Future Extensions

With the goals of implementing MCAPI efficiently, the APIs are kept simple with potential for adding more functionality on top of MCAPI later. Some specific areas include zero copy, multicast, and

informational functions for debugging, statistics (optimization) and status. These areas are strong candidates for future extensions and they are briefly described in the following subsections.

3.15.1 Zero Copy

As mentioned above zero copy can be very useful in systems where multiple nodes have shared memory. While, zero copy functionality can relatively simply be implemented on top of MCAPI providing specific API support may be useful, and something to consider for the future.

3.15.2 Multi-cast

MCAPI currently supports point-to-point communication. Communication with one sender and multiple receivers, often referred to as multi-cast, can be layered on top of MCAPI. For transports with (hardware) support for one-to-multiple communication, specific API functions supporting this type of communication, would allow for more efficient implementations than is possible through layering.

3.15.3 Debug, Statistics and Status functions

Support functions providing information for debugging, optimization and system status are useful in most systems, and would be a valuable addition to MCAPI as well, and are also worth future consideration.

3.16 MCAPI Datatypes

MCAPI uses predefined data types for maximum portability. The predefined MCAPI data types are defined in the following subsections.

3.16.1 mcapi_node_t

The `mcapi_node_t` type is used for mcapi nodes. The node numbering is implementation defined. For application portability we recommend using symbolic constants in your code.

3.16.2 mcapi_port_t

The `mcapi_port_t` type is used for mcapi ports. The port numbering is implementation defined. For application portability we recommend using symbolic constants in your code.

3.16.3 mcapi_endpoint_t

The `mcapi_endpoint_t` type is used for creating and managing endpoints for sending and receiving messages (see Section 2). It is also used to setup and to create efficient local handles for packet channels (see Section 2) and scalar channels (see Section 2). MCAPI routines for creating and managing endpoints are described in Section 4.2, packet channel creation is covered in Section 4.4.1, and scalar channel creation is covered in Section 4.5.1. The `mcapi_endpoint_t` type is an opaque datatype whose exact definition is implementation defined. The endpoint identifier is globally unique to an mcapi domain.

NOTE: The MCAPI API user should not attempt to examine the contents of this datatype as this can result in non-portable application code.

3.16.4 mcapi_pktchan_recv_hndl_t

The `mcapi_pktchan_recv_hndl_t` type is used to receive packets from a connected packet channel (see Section 2). MCAPI routines for creating and using the `mcapi_pktchan_recv_hndl_t` type are covered in Section 4.4. The `mcapi_pktchan_recv_hndl_t` is an opaque datatype whose exact definition is implementation defined.

NOTE: The MCAPI API user should not attempt to examine the contents of this datatype as this can result in non-portable application code.

3.16.5 mcapi_pktchan_send_hndl_t

The `mcapi_pktchan_send_hndl_t` type is used to send packets to a connected packet channel (see Section 2). MCAPI routines for creating and using the `mcapi_pktchan_send_hndl_t` type are covered in Section 4.4. The `mcapi_pktchan_send_hndl_t` is an opaque datatype whose exact definition is implementation defined.

NOTE: The MCAPI API user should not attempt to examine the contents of this datatype as this can result in non-portable application code.

3.16.6 mcapi_sclchan_recv_hndl_t

The `mcapi_sclchan_recv_hndl_t` type is used to receive scalars from a connected scalar channel (see Section 2). MCAPI routines for creating and using the `mcapi_sclchan_recv_hndl_t` type are covered in Section 4.5. The `mcapi_sclchan_recv_hndl_t` is an opaque datatype whose exact definition is implementation defined.

NOTE: The MCAPI API user should not attempt to examine the contents of this datatype as this can result in non-portable application code.

3.16.7 mcapi_sclchan_send_hndl_t

The `mcapi_sclchan_send_hndl_t` type is used to send scalars to a connected scalar channel (see Section 2). MCAPI routines for creating and using the `mcapi_sclchan_send_hndl_t` type are covered in Section 4.5. The `mcapi_sclchan_send_hndl_t` is an opaque datatype whose exact definition is implementation defined.

NOTE: The MCAPI API user should not attempt to examine the contents of this datatype as this can result in non-portable application code.

3.16.8 mcapi_uint64_t, mcapi_uint32_t, mcapi_uint16_t & mcapi_uint8_t,

The `mcapi_uint64_t`, `mcapi_uint32_t`, `mcapi_uint16_t`, and `mcapi_uint8_t` types are used for 64-, 32-, 16, and 8-bit scalars.

3.16.9 mcapi_request_t

The `mcapi_request_t` type is used to record the state of a pending non-blocking MCAPI transaction (see Section 3.5.4). Non-blocking MCAPI routines exist for message send and receive (see Section 4.3) and packet send and receive (see Section 4.4). The mcapi request can only be used by the node it was created on.

NOTE: The MCAPI API user should not attempt to examine the contents of this datatype as this can result in non-portable application code.

3.16.10 mcapi_status_t

The `mcapi_status_t` type is an enumerated type used to record the result of a MCAPI API call. If a status can be returned by an API call, the associated MCAPI API call will allow a `mcapi_status_t` to be passed by reference. The API call will fill in the status code and the API user may examine the CPAI_status_t variable to determine the result of the call. Implementations may use `MCAPI_ERROR` to reset the status code to a non-zero value.

3.16.11 mcapi_timeout_t

The `mcapi_timeout_t` type is a scalar type used to indicate the duration an `mcapi_wait()` or `mcapi_wait_any()` API call will block before reporting a timeout. The units of the `mcapi_timeout_t` datatype are implementation defined since mechanisms for time keeping vary from system to system. Applications should not rely on this feature for satisfaction of real time constraints as its usage will not guarantee application portability across MCAPI implementations. The `mcapi_timeout_t` datatype is intended only to allow for error detection and recovery.

4 MCAPI API

4.1 General

This section describes initialization and introspection functions.

`MCAPI_IN` and `MCAPI_OUT` are used to distinguish between input and output parameters.

4.1.1 MCAPI_INITIALIZE

NAME

`mcapi_initialize` – Initializes the MCAPI implementation.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_initialize(
     MCAPI_IN mcapi_node_t node_id,
     MCAPI_OUT mcapi_version_t* mcapi_version,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

`mcapi_initialize()` initializes the MCAPI environment on a given MCAPI node. It has to be called by each node using MCAPI. `mcapi_version` is set to the to the implementation version number. A node is a process, a thread, or a processor (or core) with an independent program counter running a piece of code. In other words, an MCAPI node is an independent thread of control. An MCAPI node can call `mcapi_initialize()` once per node, and it is an error to call `mcapi_initialize()` multiple times from a given node. A given MCAPI implementation will specify what is a node (i.e., what thread of control – process, thread, or other -- is a node) in that implementation. A thread and process are just two examples of threads of control, and there could be other.

RETURN VALUE

On success, `*mcapi_status` is set to MCAPI_SUCCESS. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

MCAPI_ENO_INIT	The MCAPI environment could not be initialized.
MCAPI_INITIALIZED	The MCAPI environment has already been initialized.
MCAPI_ENODE_NOTVALID	The parameter is not a valid node.
MCAPI_EPARAM	Incorrect `mcapi_status` or `mcapi_version` parameter.

SEE ALSO

`mcapi_finalize()`

4.1.2 MCAPI_FINALIZE

NAME

`mcapi_finalize` – Finalizes the MCAPI implementation.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_finalize(
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

`mcapi_finalize()` finalizes the MCAPI environment on a given MCAPI node. It has to be called by each node using MCAPI. It is an error to call `mcapi_finalize()` without first calling `mcapi_initialize()`. An MCAPI node can call `mcapi_finalize()` once for each call to `mcapi_initialize()`, but it is an error to call `mcapi_finalize()` multiple times from a given node unless `mcapi_initialize()` has been called prior to each `mcapi_finalize()` call.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENO_FINAL`	The MCAPI environment could not be finalized.
`MCAPI_EPARAM`	Incorrect `mcapi_status_` parameter.

SEE ALSO

`mcapi_initialize()`

4.1.3 MCAPI_GET_NODE_ID

NAME

`mcapi_get_node_id` – return the node number associated with the local node

SYNOPSIS

```
#include <mcapi.h>

mcapi_uint_t mcapi_get_node_id(
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Returns the node id associated with the local node.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENODE_NOTINIT`	The node is not initialized.
`MCAPI_EPARAM`	Incorrect `mcapi_status` parameter.

4.2 Endpoints

This section describes API functions that create, delete and modify endpoints.

4.2.1 MCAPI_CREATE_ENDPOINT

NAME

`mcapi_create_endpoint` - create an endpoint.

SYNOPSIS

```
#include <mcapi.h>


mcapi_endpoint_t mcapi_create_endpoint(
     MCAPI_IN mcapi_port_t port_id,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

`mcapi_create_endpoint()` is used to create endpoints, using the `node_id` of the local node calling the API function and specific `port_id`, returning a reference to a globally unique endpoint which can later be referenced by name using `mcapi_get_endpoint()` (see Section 4.2.3). The `port_id` can be set to `MCAPI_PORT ANY` to request the next available endpoint on the local node.

MCAPI supports a simple static naming scheme to create endpoints based on global tuple names. Other nodes can access the created endpoint by calling `mcapi_get_endpoint()` and specifying the appropriate node and port id's. Enpoints can be passed on to other endpoints and an endpoint created using `MCAPI_PORT ANY` has to be passed on to other endpoints by the creator, to facilitate communication.
Static naming allows the programmer to define an MCAPI communication topology at compile time. This facilitates simple initialization. Section 7.1 illustrates an example of initialization and bootstrapping using static naming. Creating endpoints using `MCAPI_PORT ANY` provides a convenient method to create endpoints without having to specify the `port_id`.

RETURN VALUE

On success, an endpoint is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `MCAPI_NULL` is returned and `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_EPORT_NOTVALID`	The parameter is not a valid port.
`MCAPI_EENDP_ISCREATED`	The endpoint is already created.
`MCAPI_ENODE_NOTINIT`	The node is not initialized.
`MCAPI_EENDP_LIMIT`	Exceeded maximum number of endpoints allowed.
`MCAPI_EEP_NOTALLOWED`	Endpoints cannot be created on this node.
`MCAPI_EPARAM`	Incorrect `mcapi_status` parameter.

NOTE

The node number can only be set using the `mcapi_intialize()` function.

SEE ALSO

`mcapi_initialize()`

4.2.2 MCAPI_GET_ENDPOINT_I

NAME

`mcapi_get_endpoint_i` – obtain the endpoint associated with a given tuple.

SYNOPSIS

```
#include <mcapi.h>


void mcapi_get_endpoint_i(
     MCAPI_IN mcapi_node_t node_id,
     MCAPI_IN mcapi_port_t port_id,
     MCAPI_OUT mcapi_endpoint_t* endpoint,
     MCAPI_OUT mcapi_request_t* request,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

`mcapi_get_endpoint_i()` allows other nodes (“third parties”) to get the endpoint identifier for the endpoint associated with a global tuple name <`node_id, port_id`>. This function is non-blocking and will return immediately.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_EPORT_NOTVALID`	The parameter is not a valid port.
`MCAPI_ENODE_NOTVALID`	The parameter is not a valid node.
`MCAPI_EPARAM`	Incorrect `mcapi_status` parameter.

NOTE

Use the `mcapi_test()`, `mcapi_wait()` and `mcapi_wait_any()` functions to query the status of and `mcapi_cancel()` function to cancel the operation.

SEE ALSO

`mcapi_get_node_id()`

4.2.3 MCAPI_GET_ENDPOINT

NAME

`mcapi_get_endpoint` – obtain the endpoint associated with a given tuple.

SYNOPSIS

```
#include <mcapi.h>


mcapi_endpoint_t mcapi_get_endpoint(
     MCAPI_IN mcapi_node_t node_id,
     MCAPI_IN mcapi_port_t port_id,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

`mcapi_get_endpoint()` allows other nodes (“third parties”) to get the endpoint identifier for the endpoint associated with a global tuple name <`node_id, port_id`>. This function will block until the specified remote endpoint has been created via the `mcapi_create_endpoint()` call.

RETURN VALUE

On success, an endpoint is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `MCAPI_NULL` is returned and `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_EPORT_NOTVALID`	The parameter is not a valid port.
`MCAPI_ENODE_NOTVALID`	The parameter is not a valid node.
`MCAPI_EPARAM`	Incorrect `mcapi_status` parameter.

4.2.4 MCAPI_DELETE_ENDPOINT

NAME

`mcapi_delete_endpoint` – delete an endpoint.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_delete_endpoint(
     MCAPI_IN mcapi_endpoint_t endpoint,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Deletes an MCAPI endpoint. Pending messages are discarded. If an endpoint has been connected to a packet or scalar channel, the appropriate close method must be called before deleting the endpoint. Delete is a blocking operation. Since the connection is closed before deleting the endpoint, the delete method does not require any cross-process synchronization and is guaranteed to return in a timely manner (operation will return without having to block on any IPC to any remote nodes). It is an error to attempt to delete an endpoint that has not been closed. Only the node that created an endpoint can delete it.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_ENDP`	Argument is not a valid endpoint descriptor.
`MCAPI_ECHAN_OPEN`	A channel is open, deletion is not allowed.
`MCAPI_ENOT_OWNER`	An endpoint can only be deleted by its creator.
`MCAPI_EPARAM`	Incorrect `mcapi_status` parameter.

SEE ALSO

`mcapi_create_endpoint()`

4.2.5 MCAPI_GET_ENDPOINT_ATTRIBUTE

NAME

`mcapi_get_endpoint_attribute` – get endpoint attributes.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_get_endpoint_attribute(
     MCAPI_IN mcapi_endpoint_t endpoint,
     MCAPI_IN mcapi_uint_t attribute_num,
     MCAPI_OUT void* attribute,
     MCAPI_IN size_t attribute_size,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

`mcapi_get_endpoint_attribute()` allows the programmer to query endpoint attributes related to buffer management or quality of service. `attribute_num` indicates which one of the endpoint attributes is being referenced. `attribute` points to a structure or scalar to be filled with the value of the attribute specified by `attribute_num`. `attribute size` is the size in bytes of the structure or scalar. See Section 3.2 and the example `mcapi.h` for a description of attributes. The `mcapi_get_endpoint_attribute()` function returns the requested attribute value by reference.

It is an error to attempt a connection between endpoints whose attributes are set in an incompatible way (for now, whether attributes are compatible or not is implementation defined). It is also an error to attempt to change the attributes of endpoints that are connected.

RETURN VALUE

On success, *attribute is filled with the requested attribute and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to an error code and `*attribute` is not modified.

ERRORS

`MCAPI_ENOT_ENDP`	Argument is not an endpoint descriptor.
`MCAPI_EATTR_NUM`	Unknown attribute number.
`MCAPI_EATTR_SIZE`	Incorrect attribute size.
`MCAPI_EPARAM`	Incorrect `mcapi_status` parameter.

SEE ALSO

```
mcapi_set_endpoint_attribute()
```

4.2.6 MCAPI_SET_ENDPOINT_ATTRIBUTE

NAME

`mcapi_set_endpoint_attribute` - set endpoint attributes.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_set_endpoint_attribute(
     MCAPI_IN mcapi_endpoint_t endpoint,
     MCAPI_IN mcapi_uint_t attribute_num,
     MCAPI_IN const void* attribute,
     MCAPI_IN size_t attribute_size,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

`mcapi_set_endpoint_attribute()` allows the programmer to assign endpoint attributes related to buffer management or quality of service. `attribute_num` indicates which one of the endpoint attributes is being referenced. `attribute` points to a structure or scalar to be filled with the value of the attribute specified by attribute_num. `attribute size` is the size in bytes of the structure or scalar. See Section 3.2 and `mcapi.h` for a description of attributes.

It is an error to attempt a connection between endpoints whose attributes are set in an incompatible way (for now, whether attributes are compatible or not is implementation defined). It is also an error to attempt to change the attributes of endpoints that are connected.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_ENDP` Argument is not an endpoint descriptor.

`MCAPI_EATTR_NUM` Unknown attribute number.

`MCAPI_EATTR_SIZE` Incorrect attribute size.

`MCAPI_EPARAM` Incorrect `mcapi_status` parameter.

`MCAPI_ECONNECTED` Attribute changes not allowed on connected endpoints.

`MCAPI_EREAD_ONLY` Attribute cannot be modified.

SEE ALSO

```
mcapi_get_endpoint_attribute()
```

4.3 MCAPI Messages

MCAPI Messages facilitate connectionless transfer of data buffers. The messaging API provides a "user-specified buffer" communications interface – the programmer specifies a buffer of data to be sent on the send side, and the user specifies an empty buffer to be filled with incoming data on the receive side. The implementation must be able to transfer messages to and from any buffer the programmer specifies, although the implementation may use extra buffering internally to queue up data between the sender and receiver.

4.3.1 MCAPI_MSG_SEND_I

NAME

mcapi_msg_send_i – sends a (connectionless) message from a send endpoint to a receive endpoint.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_msg_send_i(
     MCAPI_IN mcapi_endpoint_t send_endpoint,
     MCAPI_IN mcapi_endpoint_t receive_endpoint,
     MCAPI_IN void* buffer,
     MCAPI_IN size_t buffer_size,
     MCAPI_IN mcapi_priority_t priority,
     MCAPI_OUT mcapi_request_t* request,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Sends a (connectionless) message from a send endpoint to a receive endpoint. It is a non-blocking function, and returns immediately. `send_endpoint`, is a local endpoint identifying the send endpoint, `receive_endpoint` identifies a receive endpoint. `buffer` is the application provided buffer, `buffer_size` is the buffer size in bytes, `priority` determines the message priority and `request` is the identifier used to determine if the send operation has completed on the sending endpoint and the buffer can be reused by the application. Furthermore, this method will abandon the send and return `MCAPI_ENO_MEM` if the system cannot allocate enough memory at the send endpoint to queue up the outgoing message.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

MCAPI_ENOT_ENDP	Argument is not an endpoint descriptor.
MCAPI_EMESS_LIMIT	The message size exceeds the maximum size allowed by the MCAPI implementation.
MCAPI_ENO_BUFFER	No more message buffers available.
MCAPI_ENO_REQUEST	No more request handles available.
MCAPI_ENO_MEM	No memory available.
MCAPI_EPRIO	Incorrect priority level.
MCAPI_EPARAM	Incorrect `request` or `mcapi_status` parameter.

NOTE

Use the `mcapi_test()`, `mcapi_wait()` and `mcapi_wait_any()` functions to query the status of and `mcapi_cancel()` function to cancel the operation.

4.3.2 MCAPI_MSG_SEND

NAME

mcapi_msg_send – sends a (connectionless) message from a send endpoint to a receive endpoint.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_msg_send(
     MCAPI_IN mcapi_endpoint_t  send_endpoint,
     MCAPI_IN mcapi_endpoint_t  receive_endpoint,
     MCAPI_IN void* buffer,
     MCAPI_IN size_t buffer_size,
     MCAPI_IN mcapi_priority_t priority,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Sends a (connectionless) message from a send endpoint to a receive endpoint. It is a blocking function, and returns once the buffer can be reused by the application. `send_endpoint` is a local endpoint identifying the send endpoint, `receive_endpoint` identifies a receive endpoint. `buffer` is the application provided buffer and `buffer_size` is the buffer size in bytes, and `priority` determines the message priority

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below. Success means that the entire buffer has been sent.

ERRORS

MCAPI_ENOT_ENDP	Argument is not an endpoint descriptor.
MCAPI_EMESS_LIMIT	The message size exceeds the maximum size allowed by the MCAPI implementation.
MCAPI_ENO_BUFFER	No more message buffers available.
MCAPI_EPRIO	Incorrect priority level.
MCAPI_EPARAM	Incorrect `mcapi_status` parameter.

4.3.3 MCAPI_MSG_RECV_I

NAME

`mcapi_msg_recv_i` – receives a (connectionless) message from a receive endpoint.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_msg_recv_i(
     MCAPI_IN mcapi_endpoint_t  receive_endpoint,
     MCAPI_OUT void* buffer,
     MCAPI_IN size_t buffer_size,
     MCAPI_OUT mcapi_request_t* request,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Receives a (connectionless) message from a receive endpoint. It is a non-blocking function, and returns immediately. `receive_endpoint` is a local endpoint identifying the receive endpoint. `buffer` is the application provided buffer, and `buffer_size` is the buffer size in bytes. `request` is the identifier used to determine if the receive operation has completed (all the data is in the buffer).

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_ENDP`	Argument is not a valid endpoint descriptor.
`MCAPI_ETRUNCATED`	The message size exceeds the buffer_size.
`MCAPI_ENO_REQUEST`	No more request handles available.
`MCAPI_EPARAM`	Incorrect `buffer`, `request` and/or `mcapi_status` parameter.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status of and `mcapi_cancel()` function to cancel the operation.

4.3.4 MCAPI_MSG_RECV

NAME

`mcapi_msg_recv` – receives a (connectionless) message from a receive endpoint.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_msg_recv(
     MCAPI_IN mcapi_endpoint_t  receive_endpoint,
     MCAPI_OUT void* buffer,
     MCAPI_IN size_t buffer_size,
     MCAPI_OUT size_t* received_size,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Receives a (connectionless) message from a receive endpoint. It is a blocking function, and returns once a message is available and the received data filled into the buffer. `receive_endpoint` is a local endpoint identifying the receive endpoint. `buffer` is the application provided buffer, and `buffer_size` is the buffer size in bytes. The `received_size` parameter is filled with the actual size of the received message.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_ENDP`	Argument is not a valid endpoint descriptor.
`MCAPI_ETRUNCATED`	The message size exceeds the buffer_size.
`MCAPI_EPARAM`	Incorrect `buffer` and/or `mcapi_status` parameter.

4.3.5 MCAPI_MSG_AVAILABLE

NAME

`mcapi_msg_available` – checks if messages are available on a receive endpoint.

SYNOPSIS

```
#include <mcapi.h>

mcapi_uint_t mcapi_msg_available(
     MCAPI_IN mcapi_endpoint_t receive_endpoint,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Checks if messages are available on a receive endpoint. The function returns in a timely fashion. The number of "available" incoming messages is defined as the number of `mcapi_msg_recv()` operations that are guaranteed to not block waiting for incoming data. `receive_endpoint` is a local identifier for the receive endpoint. The call only checks the availability of messages and does not de-queue them. `mcapi_msg_available()` can only be used to check availability on endpoints on the node local to the caller.

RETURN VALUE

On success, the number of available messages is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `MCAPI_NULL` is returned and `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_ENDP`	Argument is not a valid endpoint descriptor.
`MCAPI_EPARAM`	Incorrect mcapi_status parameter.

NOTE

The status code must be checked to distinguish between no messages and an error condition.

4.4 MCAPI Packet Channels

MCAPI packet channels transfer data packets between a pair of connected endpoints. A connection between two endpoints is established via a two-phase process. First, some node in the system calls `mcapi_connect_pktchan_i()` to define a connection between two endpoints. This function returns immediately. In the second phase, both sender and receiver open their end of the channel by invoking `mcapi_open_pktchan_send_i()` and `mcapi_open_pktchan_recv_i()`, respectively. The connection is synchronized when both the sender and receiver open functions have completed. In order to avoid deadlock situations, the open functions are non-blocking.

This two-phased binding approach has several important benefits. The “connect” call can be made by any node in the system, which allows the programmer to define the entire channel topology in a single piece of code. This code could even be auto-generated by some static connection tool. This makes it easy to change the channel topology without having to modify multiple source files. This approach also allows the sender and receiver to do their work without any knowledge of what remote nodes they are connected to. This allows for better modularity and application scaling.

Packet channels provide a “system-specified buffer” interface. The programmer specifies the address of a buffer of data to be sent on the send side, but the receiver's recv method returns a buffer of data at an address chosen by the system. This is different from the “user-specified buffer” interface use by MCAPI messaging – with messages the programmer chooses the buffer in which data is received, and with packet channels the system chooses the buffer.

4.4.1 MCAPI_ CONNECT_PKTCHAN_I

NAME

`mcapi_connect_pktchan_i` – connects send & receive side endpoints.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_connect_pktchan_i(
     MCAPI_IN mcapi_endpoint_t  send_endpoint,
     MCAPI_IN mcapi_endpoint_t  receive_endpoint,
     MCAPI_OUT mcapi_request_t* request,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Connects a pair of endpoints into a unidirectional FIFO channel. The connect operation can be performed by the sender, the receiver, or by a third party. The connect can happen once at the start of the program, or dynamically at run time.

Connect is a non-blocking function. Synchronization to ensure the channel has been created is provided by the open call discussed later.

Attempts to make multiple connections to a single endpoint will be detected as errors. The type of channel connected to an endpoint must match the type of open call invoked by that endpoint; the open function will return an error if the opened channel type does not match the connected channel type, or if the attributes of the endpoints are incompatible.

It is an error to attempt a connection between endpoints whose attributes are set in an incompatible way (for now, whether attributes are compatible or not is implementation defined). It is also an error to attempt to change the attributes of endpoints that are connected.

RETURN VALUE

On success `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_ENDP`	Argument is not a valid endpoint descriptor.
`MCAPI_ECONNECTED`	A channel connection has already been established for one or both of the specified endpoints.
`MCAPI_ENO_REQUEST`	No more request handles available.
`MCAPI_EATTR_INCOMP`	Connection of endpoints with incompatible attributes not allowed.
`MCAPI_EPARAM`	Incorrect `request` or `mcapi_status` parameter.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status and `mcapi_cancel()` function to cancel the operation.

4.4.2 MCAPI_OPEN_PKTCHAN_RECV_I

NAME

`mcapi_open_pktchan_recv_i` – Creates a typed, local representation of the channel. It also provides synchronization for channel creation between two endpoints. Opens are required on both receive and send endpoints.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_open_pktchan_recv_i(
        MCAPI_OUT mcapi_pktchan_recv_hndl_t* recv_handle,
        MCAPI_IN mcapi_endpoint_t receive_endpoint,
        MCAPI_OUT mcapi_request_t* request,
        MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Opens the receive end of a packet channel. The corresponding calls are required on both sides for synchronization to ensure that the channel has been created. It is a non-blocking function, and the `recv_handle` is filled in upon successful completion. No specific ordering of calls between sender and receiver is required since the call is non-blocking. `receive_endpoint` is the endpoint associated with the channel. The open call returns a typed, local handle for the connected channel that is used for channel receive operations.

RETURN VALUE

On success, a valid request is returned by and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_ENDP`	Argument is not a valid endpoint descriptor.
`MCAPI_ENOT_CONNECTED`	The channel is not connected (cannot be opened).
`MCAPI_ECHAN_TYPE`	Attempt to open a packet channel on an endpoint that has been connected with a different channel type.
`MCAPI_EDIR`	Attempt to open a send handle on a port that was connected as a receiver, or vice versa.
`MCAPI_EPARAM`	Incorrect `request` or `mcapi_status` parameter.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status and `mcapi_cancel()` function to cancel the operation.

4.4.3 MCAPI_OPEN_PKTCHAN_SEND_I

NAME

`mcapi_open_pktchan_send_i` – Creates a typed, local representation of the channel. It also provides synchronization for channel creation between two endpoints. Opens are required on both receive and send endpoints.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_open_pktchan_send_i(
        MCAPI_OUT mcapi_pktchan_send_hndl_t* send_handle,
        MCAPI_IN mcapi_endpoint_t  send_endpoint,
        MCAPI_OUT mcapi_request_t* request,
        MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Opens the send end of a packet channel. The corresponding calls are required on both sides for synchronization to ensure that the channel has been created. It is a non-blocking function, and the `send_handle` is filled in upon successful completion. No specific ordering of calls between sender and receiver is required since the call is non-blocking. `send_endpoint` is the endpoint associated with the channel. The open call returns a typed, local handle for the connected endpoint that is used by channel send operations.

RETURN VALUE

On success, a valid `request` is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_ENDP`	Argument is not a valid endpoint descriptor.
`MCAPI_ECHAN_TYPE`	Attempt to open a packet channel on an endpoint that has been connected with a different channel type.
`MCAPI_EDIR`	Attempt to open a send handle on a port that was connected as a receiver, or vice versa.
`MCAPI_EPARAM`	Incorrect `request` or `mcapi_status` parameter.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status and `mcapi_cancel()` function to cancel the operation.

4.4.4 MCAPI_PKTCHAN_SEND_I

NAME

`mcapi_pktchan_send_i` – sends a (connected) packet on a channel.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_pktchan_send_i(
     MCAPI_IN mcapi_pktchan_send_hndl_t send_handle,
     MCAPI_IN void* buffer,
     MCAPI_IN size_t size,
     MCAPI_OUT mcapi_request_t* request,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Sends a packet on a connected channel. It is a non-blocking function, and returns immediately. `buffer` is the application provided buffer and `size` is the buffer size. `request` is the identifier used to determine if the send operation has completed on the sending endpoint and the buffer can be reused. While this method returns immediately, data transfer will not complete until there is sufficient free space in the channels receive buffer. A subsequent call to `mcapi_wait()` will block until space becomes available at the receiver, the send operation has completed, and the send buffer is available for reuse. Furthermore, this method will abandon the send and return `MCAPI_ENO_MEM` if the system cannot allocate enough memory at the send endpoint to queue up the outgoing packet.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_HANDLE`	Argument is not a channel handle.
`MCAPI_EPACK_LIMIT`	The packet size exceeds the maximum size allowed by the MCAPI implementation.
`MCAPI_ENO_BUFFER`	No more packet buffers available.
`MCAPI_ENO_REQUEST`	No more request handles available.
`MCAPI_ENO_MEM`	No memory available.
`MCAPI_EPARAM`	Incorrect `request` or `mcapi_status` parameter.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status and `mcapi_cancel()` function to cancel the operation.

4.4.5 MCAPI_PKTCHAN_SEND

NAME

`mcapi_pktchan_send` – sends a (connected) packet on a channel.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_pktchan_send(
     MCAPI_IN mcapi_pktchan_send_hndl_t send_handle,
     MCAPI_IN void* buffer,
     MCAPI_IN size_t size,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Sends a packet on a connected channel. It is a blocking function, and returns once the buffer can be reused. `send_handle` is the efficient local send handle which represents the send endpoint associated with the channel. `buffer` is the application provided buffer and `size` is the buffer size. Since channels behave like FIFOs, this method will block if there is no free space in the channel's receive buffer. When sufficient space becomes available (due to receive calls), the function will complete.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below. Success means that the entire buffer has been sent.

ERRORS

`MCAPI_ENOT_HANDLE`	Argument is not a channel handle.
`MCAPI_EPACK_LIMIT`	The message size exceeds the maximum size allowed by the MCAPI implementation.
`MCAPI_ENO_BUFFER`	No more packet buffers available.
`MCAPI_EPARAM`	Incorrect `mcapi_status` parameter.

4.4.6 MCAPI_PKTCHAN_RECV_I

NAME

`mcapi_pktchan_recv_i` – receives a (connected) packet on a channel.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_pktchan_recv_i(
     MCAPI_IN mcapi_pktchan_recv_hndl_t receive_handle,
     MCAPI_OUT void** buffer,
     MCAPI_OUT mcapi_request_t* request,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Receives a packet on a connected channel. It is a non-blocking function, and returns immediately. `receive_handle` is the receive endpoint. At some point in the future, when the receive operation completes, the `buffer` parameter is filled with the address of a system-supplied buffer containing the received packet. After the receive request has completed and the application is finished with `buffer`, `buffer` should be returned to the system by calling `mcapi_pktchan_free()`. `request` is the identifier used to determine if the receive operation has completed and `buffer` is ready for use; the `mcapi_test()` , `mcapi_wait()` or `mcapi_wait_any()` function will return the actual size of the received packet.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_HANDLE`	Argument is not a channel handle.
`MCAPI_EPACKLIMIT`	The packet size exceeds the maximum size allowed by the MCAPI implementation.
`MCAPI_ENO_BUFFER`	No more packet buffers available.
`MCAPI_ENO_REQUEST`	No more request handles available.
`MCAPI_EPARAM`	Incorrect `buffer`, `request` and/or `mcapi_status` parameter.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status of and `mcapi_cancel()` function to cancel the operation.

4.4.7 MCAPI_PKTCHAN_RECV

NAME

`mcapi_pktchan_recv` – receives a data packet on a (connected) channel.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_pktchan_recv(
     MCAPI_IN mcapi_pktchan_recv_hndl_t receive_handle,
     MCAPI_OUT void** buffer,
     MCAPI_OUT size_t* received_size,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Receives a packet on a connected channel. It is a blocking function, and returns when the data has been written to the buffer. `receive_handle` is the efficient local representation of the receive endpoint associated with the channel. `buffer` is filled with a pointer to the system-supplied receive buffer and `received_size` is filled with the size of the packet in that buffer. When the application finishes with buffer, it must return it to the system by calling `mcapi_pktchan_free()`.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_HANDLE`	Argument is not a channel handle.
`MCAPI_EPACKLIMIT`	The package size exceeds the maximum size allowed by the MCAPI implementation.
`MCAPI_ENO_BUFFER`	No more packet buffers available.
`MCAPI_EPARAM`	Incorrect `buffer` and/or `mcapi_status` parameter.

4.4.8 MCAPI_PKTCHAN_AVAILABLE

NAME

`mcapi_pktchan_available` – checks if packets are available on a receive endpoint.

SYNOPSIS

```
#include <mcapi.h>

mcapi_uint_t mcapi_pktchan_available(
     MCAPI_IN mcapi_pktchan_recv_hndl_t receive_handle,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Checks if packets are available on a receive endpoint. This function returns in a timely fashion. The number of available packets is defined as the number of receive operations that could be performed without blocking to wait for incoming data. `receive_handle` is the efficient local handle for the packet channel. The call only checks the availability of messages and does not de-queue them.

RETURN VALUE

On success, the number of available packets are returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `MCAPI_NULL` is returned and `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_HANDLE`	Argument is not a channel handle.
`MCAPI_EPARAM`	Incorrect `mcapi_status` parameter.

NOTE

The status code must be checked to distinguish between no messages and an error condition.

4.4.9 MCAPI_PKTCHAN_FREE

NAME

mcapi_pktchan_free – releases a packet buffer obtained from a mcapi_pktchan_recv() call.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_pktchan_free(
     MCAPI_IN void* buffer,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

When a user is finished with a packet buffer obtained from mcapi_pktchan_recv_i() or mcapi_pktchan_recv(), they should invoke this function to return the buffer to the system. Buffers can be freed in any order. This function is guaranteed to return in a timely fashion.

RETURN VALUE

On success *mcapi_status is set to MCAPI_SUCCESS. On error, *mcapi_status is set to the appropriate error defined below.

ERRORS

MCAPI_ENOT_VALID_BUF	Argument is not a valid buffer descriptor.
MCAPI_EPARAM	Incorrect mcapi_status parameter.

SEE ALSO

mcapi_pktchan_recv(), mcapi_pktchan_recv_i()

4.4.10 MCAPI_ PACKETCHAN_RECV_CLOSE_I

NAME

`mcapi_packetchan_recv_close_i` – closes channel on a receive endpoint.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_pktchan_recv_close_i(
     MCAPI_IN mcapi_pktchan_recv_hndl_t receive_handle,
     MCAPI_OUT mcapi_request_t* request,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Closes the receive side of a channel. The sender makes the send-side call and the receiver makes the receive-side call. The corresponding calls are required on both sides to ensure that the channel has been properly closed. It is a non-blocking function, and returns immediately. `receive_handle` is the receive endpoint identifier. All pending packets are discarded, and any attempt to send more packets will give an error.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_HANDLE`	Argument is not a channel handle.
`MCAPI_ENOT_OPEN`	The endpoint is not open.
`MCAPI_EPARAM`	Incorrect `request` or `mcapi_status` parameter.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status of and `mcapi_cancel()` function to cancel the operation.

4.4.11 MCAPI_ PACKETCHAN_SEND_CLOSE_I

NAME

`mcapi_pktchan_send_close_i` – closes channel on a send endpoint.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_pktchan_send_close_i(
     MCAPI_IN mcapi_pktchan_send_hndl_t send_handle,
     MCAPI_OUT mcapi_request_t* request,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Closes the send side of a channel. The sender makes the send-side call and the receiver makes the receive-side call. The corresponding calls are required on both sides to ensure that the channel has been properly closed. It is a non-blocking function, and returns immediately. `send_handle` is the send endpoint identifier. Pending packets at the receiver are not discarded.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_HANDLE`	Argument is not a channel handle.
`MCAPI_ENOT_OPEN`	The endpoint is not open.
`MCAPI_EPARAM`	Incorrect `request` or `mcapi_status` parameter.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status of and `mcapi_cancel()` function to cancel the operation.

4.5 MCAPI Scalar Channels

MCAPI scalar channels are used to transfer scalars, 8-bit, 16-bit, 32-bit and 64-bit scalars on a connected channel. The connection process for scalar channels uses the same two-phase mechanism as packet channels. See the packet channel section for a detailed description of the connection process.

The MCAPI scalar channels API provides only blocking send and receive methods. Scalar channels are intended to provide a very low overhead interface for moving a stream of values. In fact, some embedded systems may be able to implement a scalar channel as a hardware FIFO. The sort of streaming algorithms that take advantage of scalar channels should not require a non-blocking send or receive method; each process should simply receive a value to work on, do its work, send the result out on a channel, and repeat. Applications that require non-blocking semantics should use packet channels instead of scalar channels. The scalar functions only support communication of same size scalars on both sides, i.e. an 8-bit send must be read by an 8-bit receive.

4.5.1 MCAPI_CONNECT_SCLCHAN_I

NAME

`mcapi_connect_sclchan_i` – connects a pair of scalar channel endpoints.

SYNOPSIS

```
#include <mcapi.h>

void  mcapi_connect_sclchan_i(
     MCAPI_IN mcapi_endpoint_t send_endpoint,
     MCAPI_IN mcapi_endpoint_t receive_endpoint,
     MCAPI_OUT mcapi_request_t* request,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Connects a pair of endpoints. The connect operation can be performed by the sender, the receiver, or by a third party. The connect can happen once at the start of the program or dynamically at run time.

`mcapi_connect_sclchan_i()` is a non-blocking function. Synchronization to ensure the channel has been created is provided by the open call discussed later.

Note that this function behaves like the packetchannel connect call.

Attempts to make multiple connections to a single endpoint will be detected as errors. The type of channel connected to an endpoint must match the type of open call invoked by that endpoint; the open function will return an error if the opened channel type does not match the connected channel type, or if the attributes of the endpoints are incompatible.

It is an error to attempt a connection between endpoints whose attributes are set in an incompatible way (for now, whether attributes are compatible or not is implementation defined). It is also an error to attempt to change the attributes of endpoints that are connected.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_ENDP`	Argument is not a valid endpoint descriptor.
`MCAPI_ECONNECTED`	A channel connection has already been established for one or both of the specified endpoints.
`MCAPI_EATTR_INCOMP`	Connection of endpoints with incompatible attributes not allowed.
`MCAPI_ENO_REQUEST`	No more request handles available.

MCAPI_EPARAM	Incorrect `request` or `mcapi_status` parameter.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status of and `mcapi_cancel()` function to cancel the operation.

4.5.2 MCAPI_OPEN_SCLCHAN_RECV_I

NAME

`mcapi_open_sclchan_recv_i` – Creates a typed, local representation of a scalar channel.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_open_sclchan_recv_i(
        MCAPI_OUT mcapi_sclchan_recv_hndl_t* receive_handle,
        MCAPI_IN mcapi_endpoint_t receive_endpoint,
        MCAPI_OUT mcapi_request_t* request,
        MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Opens the receive end of a scalar channel. It also provides synchronization for channel creation between two endpoints. The corresponding calls are required on both sides to synchronize the endpoints. It is a non-blocking function, and the `recv_handle` is filled in upon successful completion. No specific ordering of calls between sender and receiver is required since the call is non-blocking. `receive_endpoint` is the local endpoint identifier. The call returns a local handle for the connected channel.

RETURN VALUE

On success, a channel handle is returned by reference and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_ENDP`	Argument is not an endpoint descriptor.
`MCAPI_ENOT_CONNECTED`	The channel is not connected (cannot be opened).
`MCAPI_ECHAN_TYPE`	Attempt to open a packet channel on an endpoint that has been connected with a different channel type.
`MCAPI_EDIR`	Attempt to open a send handle on a port that was connected as a receiver, or vice versa.
`MCAPI_EPARAM`	Incorrect `request` or `mcapi_status` parameter.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status and `mcapi_cancel()` function to cancel the operation.

4.5.3 MCAPI_OPEN_SCLCHAN_SEND_I

NAME

`mcapi_open_sclchan_send_i` – Creates a typed, local representation of a scalar channel.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_open_sclchan_send_i(
        MCAPI_OUT mcapi_sclchan_send_hndl_t* send_handle,
        MCAPI_IN mcapi_endpoint_t send_endpoint,
        MCAPI_OUT mcapi_request_t* request,
        MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Opens the send end of a scalar channel. . It also provides synchronization for channel creation between two endpoints. The corresponding calls are required on both sides to synchronize the endpoints. It is a non-blocking function, and the `send_handle` is filled in upon successful completion. No specific ordering of calls between sender and receiver is required since the call is non-blocking. `send_endpoint` is the local endpoint identifier. The call returns a local handle for connected channel.

RETURN VALUE

On success, a channel handle is returned by reference and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_ENDP`	Argument is not an endpoint descriptor.
`MCAPI_ECHAN_TYPE`	Attempt to open a packet channel on an endpoint that has been connected with a different channel type.
`MCAPI_EDIR`	Attempt to open a send handle on a port that was connected as a receiver, or vice versa.
`MCAPI_EPARAM`	Incorrect `request` or `mcapi_status` parameter.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status and `mcapi_cancel()` function to cancel the operation.

4.5.4 MCAPI_SCLCHAN_SEND_UINT64

NAME

`mcapi_sclchan_send_uint64` – sends a (connected) 64-bit scalar on a channel.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_sclchan_send_uint64(
     MCAPI_IN mcapi_sclchan_send_hndl_t send_handle,
     MCAPI_IN mcapi_uint64_t dataword,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Sends a scalar on a connected channel. It is a blocking function, and returns immediately unless the buffer is full. `send_handle` is the send endpoint identifier. `dataword` is the scalar. Since channels behave like FIFOs, this method will block if there is no free space in the channel's receive buffer. When sufficient space becomes available (due to receive calls), the function will complete.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error `*mcapi_status` is set to the appropriate error defined below. Optionally, implementations may choose to always set `*mcapi_status` to `MCAPI_SUCCESS` for performance reasons.

ERRORS

`MCAPI_ENOT_HANDLE`	Argument is not a channel handle.
`MCAPI_EPARAM`	Incorrect `mcapi_status` parameter.

4.5.5 MCAPI_SCLCHAN_SEND_UINT32

NAME

`mcapi_sclchan_send_uint32` – sends a (connected) 32-bit scalar on a channel.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_sclchan_send_uint32(
     MCAPI_IN mcapi_sclchan_send_hndl_t send_handle,
     MCAPI_IN mcapi_uint32_t dataword,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Sends a scalar on a connected channel. It is a blocking function, and returns immediately unless the buffer is full. `send_handle` is the send endpoint identifier. `dataword` is the scalar. Since channels behave like FIFOs, this method will block if there is no free space in the channel's receive buffer. When sufficient space becomes available (due to receive calls), the function will complete.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error `*mcapi_status` is set to the appropriate error defined below. Optionally, implementations may choose to always set `*mcapi_status` to `MCAPI_SUCCESS` for performance reasons.

ERRORS

`MCAPI_ENOT_HANDLE`	Argument is not a channel handle.
`MCAPI_EPARAM`	Incorrect `mcapi_status` parameter.

4.5.6 MCAPI_SCLCHAN_SEND_UINT16

NAME

`mcapi_sclchan_send_uint16` – sends a (connected) 16-bit scalar on a channel.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_sclchan_send_uint16(
     MCAPI_IN mcapi_sclchan_send_hndl_t send_handle,
     MCAPI_IN mcapi_uint16_t dataword,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Sends a scalar on a connected channel. It is a blocking function, and returns immediately unless the buffer is full. `send_handle` is the send endpoint identifier. `dataword` is the scalar. Since channels behave like FIFOs, this method will block if there is no free space in the channel's receive buffer. When sufficient space becomes available (due to receive calls), the function will complete.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error `*mcapi_status` is set to the appropriate error defined below. Optionally, implementations may choose to always set `*mcapi_status` to `MCAPI_SUCCESS` for performance reasons.

ERRORS

`MCAPI_ENOT_HANDLE`	Argument is not a channel handle.
`MCAPI_EPARAM`	Incorrect `mcapi_status` parameter.

4.5.7 MCAPI_SCLCHAN_SEND_UINT8

NAME

`mcapi_sclchan_send_uint8` – sends a (connected) 8-bit scalar on a channel.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_sclchan_send_uint8(
     MCAPI_IN mcapi_sclchan_send_hndl_t send_handle,
     MCAPI_IN mcapi_uint8_t dataword,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Sends a scalar on a connected channel. It is a blocking function, and returns immediately unless the buffer is full. `send_handle` is the send endpoint identifier. `dataword` is the scalar. Since channels behave like FIFOs, this method will block if there is no free space in the channel's receive buffer. When sufficient space becomes available (due to receive calls), the function will complete.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error `*mcapi_status` is set to the appropriate error defined below. Optionally, implementations may choose to always set `*mcapi_status` to `MCAPI_SUCCESS` for performance reasons.

ERRORS

`MCAPI_ENOT_HANDLE`	Argument is not a channel handle.
`MCAPI_EPARAM`	Incorrect `mcapi_status` parameter.

4.5.8 MCAPI_SCLCHAN_RECV_UINT64

NAME

`mcapi_sclchan_recv_uint64` – receives a (connected) 64-bit scalar on a channel.

SYNOPSIS

```
#include <mcapi.h>

mcapi_uint64_t mcapi_sclchan_recv_uint64(
     MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Receives a scalar on a connected channel. It is a blocking function, and returns when a scalar is available. `receive_handle` is the receive endpoint identifier.

RETURN VALUE

On success, a value of type `uint64_t` is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, the return value is undefined and `*mcapi_status` is set to the appropriate error defined below. Optionally, implementations may choose to always set `*mcapi_status` to `MCAPI_SUCCESS` for performance reasons.

ERRORS

`MCAPI_ENOT_HANDLE`	Argument is not a channel handle.
`MCAPI_EPARAM`	Incorrect mcapi_status parameter.
`MCAPI_ESCL_SIZE`	Incorrect scalar size.
`MCAPI_EPARAM`	Incorrect `mcapi_status` parameter.

NOTE

The receive scalar size must match the send size.

4.5.9 MCAPI_SCLCHAN_RECV_UINT32

NAME

`mcapi_sclchan_recv_uint32` – receives a 32-bit scalar on a (connected) channel.

SYNOPSIS

```
#include <mcapi.h>

mcapi_uint32_t mcapi_sclchan_recv_uint32(
     MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Receives a scalar on a connected channel. It is a blocking function, and returns when a scalar is available. `receive_handle` is the receive endpoint identifier.

RETURN VALUE

On success, a value of type `uint32_t` is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, the return value is undefined and `*mcapi_status` is set to the appropriate error defined below. Optionally, implementations may choose to always set `*mcapi_status` to `MCAPI_SUCCESS` for performance reasons.

ERRORS

`MCAPI_ENOT_HANDLE`	Argument is not a channel handle.
`MCAPI_EPARAM`	Incorrect mcapi_status parameter.
`MCAPI_ESCL_SIZE`	Incorrect scalar size.
`MCAPI_EPARAM`	Incorrect `mcapi_status` parameter.

NOTE

The receive scalar size must match the send size.

4.5.10 MCAPI_ SCLCHAN_RECV_UINT16

NAME

`mcapi_sclchan_recv_uint16` – receives a 16-bit scalar on a (connected) channel.

SYNOPSIS

```
#include <mcapi.h>

mcapi_uint16_t mcapi_sclchan_recv_uint16(
     MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Receives a scalar on a connected channel. It is a blocking function, and returns when a scalar is available. `receive_handle` is the receive endpoint identifier.

RETURN VALUE

On success, a value of type `uint16_t` is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, the return value is undefined and `*mcapi_status` is set to the appropriate error defined below. Optionally, implementations may choose to always set `*mcapi_status` to `MCAPI_SUCCESS` for performance reasons.

ERRORS

`MCAPI_ENOT_HANDLE`	Argument is not a channel handle.
`MCAPI_EPARAM`	Incorrect mcapi_status parameter.
`MCAPI_ESCL_SIZE`	Incorrect scalar size.
`MCAPI_EPARAM`	Incorrect `mcapi_status` parameter.

NOTE

The receive scalar size must match the send size.

4.5.11 MCAPI_SCLCHAN_RECV_UINT8

NAME

`mcapi_sclchan_recv_uint8` – receives a (connected) 8-bit scalar on a channel.

SYNOPSIS

```
#include <mcapi.h>

mcapi_uint8_t mcapi_sclchan_recv_uint8(
     MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Receives a scalar on a connected channel. It is a blocking function, and returns when a scalar is available. `receive_handle` is the receive endpoint identifier.

RETURN VALUE

On success, a value of type `uint8_t` is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, the return value is undefined and `*mcapi_status` is set to the appropriate error defined below. Optionally, implementations may choose to always set `*mcapi_status` to `MCAPI_SUCCESS` for performance reasons.

ERRORS

`MCAPI_ENOT_HANDLE`	Argument is not a channel handle.
`MCAPI_EPARAM`	Incorrect mcapi_status parameter.
`MCAPI_ESCL_SIZE`	Incorrect scalar size.
`MCAPI_EPARAM`	Incorrect `mcapi_status` parameter.

NOTE

The receive scalar size must match the send size.

4.5.12 MCAPI_SCLCHAN_AVAILABLE

NAME

`mcapi_sclchan_available` – checks if scalars are available on a receive endpoint.

SYNOPSIS

```
#include <mcapi.h>

mcapi_uint_t mcapi_sclchan_available (
     MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Checks if scalars are available on a receive endpoint. The function returns immediately. `receive_endpoint` is the receive endpoint identifier. The call only checks the availability of messages does not de-queue them.

RETURN VALUE

On success, the number of available scalars are returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `MCAPI_NULL` is returned and `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_HANDLE`	Argument is not a channel handle.
`MCAPI_EPARAM`	Incorrect `mcapi_status` parameter.

NOTE

The status code must be checked to distinguish between no messages and an error condition.

4.5.13 MCAPI_ SCLCHAN_RECV_CLOSE_I

NAME

`mcapi_ sclchan_recv_close_i` – closes channel on a receive endpoint.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_sclchan_recv_close_i(
     MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
     MCAPI_OUT mcapi_request_t* request,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Closes the receive side of a channel. The corresponding calls are required on both send and receive sides to ensure that the channel is properly closed. It is a non-blocking function, and returns immediately. `receive_handle` is the receive endpoint identifier. All pending scalars are discarded, and any attempt to send more scalars will give an error.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_HANDLE`	Argument is not a channel handle.
`MCAPI_ENOT_OPEN`	The endpoint is not open.
`MCAPI_EPARAM`	Incorrect `request` or `mcapi_status` parameter.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status of and `mcapi_cancel()` function to cancel the operation.

4.5.14 MCAPI_ SCLCHAN_SEND_CLOSE_I

NAME

`mcapi_sclchan_send_close_i` – closes channel on a send endpoint.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_sclchan_send_close_i(
        MCAPI_IN mcapi_sclchan_send_hndl_t send_handle,
        MCAPI_OUT mcapi_request_t* request,
        MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Closes the send side of a channel. The corresponding calls are required on both send and receive sides to ensure that the channel is properly closed. It is a non-blocking function, and returns immediately. `send_handle` is the send endpoint identifier. Pending scalars at the receiver are not discarded.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOT_HANDLE`	Argument is not a channel handle.
`MCAPI_ENOT_OPEN`	The endpoint is not open.
`MCAPI_EPARAM`	Incorrect `request` or `mcapi_status` parameter.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status of and `mcapi_cancel()` function to cancel the operation.

4.6 MCAPI non-blocking operations

The connectionless message and packet channel API functions have both blocking and non-blocking variants. The non-blocking versions fill in a `mcapi_request_t` object and return control to the user before the communication operation is completed. The user can then use the `mcapi_test()`, `mcapi_wait()`, and `mcapi_wait_any()` functions to query the status of the non-blocking operation.

4.6.1 MCAPI_TEST

NAME

`mcapi_test` – tests if non-blocking operation has completed.

SYNOPSIS

```
#include <mcapi.h>

mcapi_boolean_t mcapi_test(
     MCAPI_IN mcapi_request_t* request,
     MCAPI_OUT size_t* size,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Checks if a non-blocking operation has completed. The function returns in a timely fashion. `request` is the identifier for the non-blocking operation. The call only checks the completion of an operation and doesn't affect any messages/packets/scalars. If the specified request completes and the pending operation was a send or receive operation, the `size` parameter is set to the number of bytes that were either sent or received by the non-blocking transaction.

RETURN VALUE

On success, `MCAPI_TRUE` is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. If the operation has not completed `MCAPI_FALSE` is returned and `*mcapi_status` is set to `MCAPI_INCOMPLETE`. On error `MCAPI_FALSE` is returned and `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOTREQ_HANDLE`	Argument is not a valid request handle.
`MCAPI_EPARAM`	Incorrect size and/or mcapi_status parameter.

SEE ALSO

```
mcapi_endpoint_t mcapi_get_endpoint_i(), mcapi_msg_send_i(),
mcapi_msg_recv_i(), mcapi_connect_pktchan_i(),
mcapi_open_pktchan_recv_i(), mcapi_open_pktchan_send_i(),
mcapi_pktchan_send_i(), mcapi_pktchan_recv_i(),
mcapi_pktchan_recv_close_i(), mcapi_pktchan_send_close_i(),
mcapi_connect_sclchan_i(), mcapi_open_sclchan_recv_i(),
mcapi_open_sclchan_send_i(), mcapi_sclchan_recv_close_i(),
mcapi_sclchan_send_close_i()
```

4.6.2 MCAPI_WAIT

NAME

mcapi_wait – waits for a non-blocking operation to complete.

SYNOPSIS

```
#include <mcapi.h>

mcapi_boolean_t mcapi_wait(
     MCAPI_IN mcapi_request_t* request,
     MCAPI_OUT size_t* size,
     MCAPI_OUT mcapi_status_t* mcapi_status,
     MCAPI_IN mcapi_timeout_t timeout
);
```

DESCRIPTION

Wait until a non-blocking operation has completed. It is a blocking function and returns when the operation has completed, has been canceled, or a timeout has occurred. `request` is the identifier for the non-blocking operation. The call only waits for the completion of an operation (all buffers referenced in the operation have been filled or consumed and can now be safely accessed by the application) and doesn't affect any messages/packets/scalars. The `size` parameter is set to number of bytes that were either sent or received by the non-blocking transaction that completed (size is irrelevant for non-blocking connect and close calls). The `mcapi_wait()` call will return if the request is cancelled by a call to `mcapi_cancel()`, and the returned `mcapi_status` will indicate that the request was cancelled. The units for `timeout` are implementation defined. If a timeout occurs the returned status will indicate that the timeout occurred. A value of `MCAPI_INFINITE` for the `timeout` parameter indicates no timeout is requested.

RETURN VALUE

On success, `MCAPI_TRUE` is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error `MCAPI_FALSE` is returned and `*mcapi_status` is set to the appropriate error defined below.

ERRORS

MCAPI_ENOTREQ_HANDLE	Argument is not a valid request handle.
MCAPI_EREQ_CANCELED	The request was canceled, by another thread (during the waiting).
MCAPI_EREQ_TIMEOUT	The operation timed out.
MCAPI_EPARAM	Incorrect `size` and/or `mcapi_status` parameter.

SEE ALSO

```
mcapi_endpoint_t mcapi_get_endpoint_i(), mcapi_msg_send_i(),
mcapi_msg_recv_i(), mcapi_connect_pktchan_i(),
mcapi_open_pktchan_recv_i(), mcapi_open_pktchan_send_i(),
mcapi_pktchan_send_i(), mcapi_pktchan_recv_i(),
```

```
mcapi_pktchan_recv_close_i(), mcapi_pktchan_send_close_i(),
mcapi_connect_sclchan_i(), mcapi_open_sclchan_recv_i(),
mcapi_open_sclchan_send_i(), mcapi_sclchan_recv_close_i(),
mcapi_sclchan_send_close_i()
```

4.6.3 MCAPI_WAIT_ANY

NAME

mcapi_wait_any – waits for any non-blocking operation in a list to complete.

SYNOPSIS

```
#include <mcapi.h>

mcapi_int_t mcapi_wait_any(
     MCAPI_IN size_t number,
     MCAPI_IN mcapi_request_t** requests,
     MCAPI_OUT size_t* size,
     MCAPI_OUT mcapi_status_t* mcapi_status,
     MCAPI_IN mcapi_timeout_t timeout
);
```

DESCRIPTION

Wait until any non-blocking operation of a list has completed. It is a blocking function and returns the index into the `requests` array (starting from 0) indicating which of any outstanding operations has completed. `number` is the number of requests in the array. `requests` is the array of `mcapi_request_t` identifiers for the non-blocking operations. The call only waits for the completion of an operation and doesn't affect any messages/packets/scalars. The `size` parameter is set to number of bytes that were either sent or received by the non-blocking transaction that completed (`size` is irrelevant for non-blocking connect and close calls). The `mcapi_wait_any()` call will return 0 if all the requests are cancelled by calls to `mcapi_cancel()` (during the waiting). The returned status will indicate that a request was cancelled. The units for `timeout` are implementation defined. If a timeout occurs the mcapi_`status` parameter will indicate that a timeout occurred. A value of `MCAPI_INFINITE` for the `timeout` parameter indicates no timeout is requested.

RETURN VALUE

On success, the index into the `requests` array of the `mcapi_request_t` identifier that has completed or has been canceled is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error MCAPI_NULL is returned and `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOTREQ_HANDLE`	Argument is not a valid request handle.
`MCAPI_EREQ_CANCELED`	One of the requests was canceled, by another thread (during the waiting).
`MCAPI_EREQ_TIMEOUT`	The operation timed out.
`MCAPI_EPARAM`	Incorrect `requests`, `size`, and/or `mcapi_status` parameter.

SEE ALSO

```
mcapi_endpoint_t mcapi_get_endpoint_i(), mcapi_msg_send_i(),
mcapi_msg_recv_i(), mcapi_connect_pktchan_i(),
mcapi_open_pktchan_recv_i(), mcapi_open_pktchan_send_i(),
mcapi_pktchan_send_i(), mcapi_pktchan_recv_i(),
mcapi_pktchan_recv_close_i(), mcapi_pktchan_send_close_i(),
mcapi_connect_sclchan_i(), mcapi_open_sclchan_recv_i(),
mcapi_open_sclchan_send_i(), mcapi_sclchan_recv_close_i(),
mcapi_sclchan_send_close_i()
```

4.6.4 MCAPI_CANCEL

NAME

`mcapi_cancel` – cancels an outstanding non-blocking operation.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_cancel(
     MCAPI_IN mcapi_request_t* request,
     MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Cancels an outstanding non-blocking operation. It is a blocking function and returns when the operation has been canceled. `request` is the identifier for the non-blocking operation. Any pending calls to `mcapi_wait()` or `mcapi_wait_any()` for this request will also be cancelled. The returned status of a canceled `mcapi_wait()` or `mcapi_wait_any()` call will indicate that the request was cancelled.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ENOTREQ_HANDLE`	Argument is not a valid request handle (the operation may have completed).
`MCAPI_EPARAM`	Incorrect mcapi_status parameter.

SEE ALSO

```
mcapi_endpoint_t mcapi_get_endpoint_i(), mcapi_msg_send_i(),
mcapi_msg_recv_i(), mcapi_connect_pktchan_i(),
mcapi_open_pktchan_recv_i(), mcapi_open_pktchan_send_i(),
mcapi_pktchan_send_i(), mcapi_pktchan_recv_i(),
mcapi_pktchan_recv_close_i(), mcapi_pktchan_send_close_i(),
mcapi_connect_sclchan_i(), mcapi_open_sclchan_recv_i(),
mcapi_open_sclchan_send_i(), mcapi_sclchan_recv_close_i(),
mcapi_sclchan_send_close_i()
```

5 Example mcapi.h file

The mcapi.h header file below is an example providing an illustration of what a mcapi.h could look like; **it may be incomplete and does NOT suggest or imply particular implementation**.

```
/*
 * mcapi.h
*/

#ifndef MCAPI_H
#define MCAPI_H

#include <stddef.h>           /* Required for size_t */

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

/*
 * MCAPI type definitions
 */

typedef int                   mcapi_int_t;
typedef unsigned int          mcapi_uint_t;
typedef unsigned char         mcapi_uint8_t;
typedef unsigned short        mcapi_uint16_t;
typedef unsigned long         mcapi_uint32_t;
typedef unsigned long long    mcapi_uint64_t;
typedef unsigned char         mcapi_boolean_t;
typedef unsigned int          mcapi_endpoint_t;
typedef unsigned int          mcapi_node_t;
typedef int                   mcapi_port_t;
typedef unsigned int          mcapi_version_t;
typedef int                   mcapi_status_t;
typedef unsigned int          mcapi_request_t;
typedef unsigned int          mcapi_priority_t;
typedef int                   mcapi_timeout_t;
typedef unsigned int          mcapi_pktchan_recv_hndl_t;
typedef unsigned int          mcapi_pktchan_send_hndl_t;
typedef unsigned int          mcapi_sclchan_send_hndl_t;
typedef unsigned int          mcapi_sclchan_recv_hndl_t;

/*
 * MCAPI Attribute Buffer Types
 */
enum mcapi_buffer_type {
      MCAPI_FIFO_BUFFER
};

/*
 * MCAPI Attribute Memory Types
 */
enum mcapi_memory_type {
      MCAPI_SHARED_MEMORY,
      MCAPI_LOCAL_MEMORY,
      MCAPI_REMOTE_MEMORY
};

/*
 * MCAPI Constants, the MAX values are implementation defined
```

```
 */
#define MCAPI_MAX_NODES             64    /* Max number of nodes */
#define MCAPI_MAX_ENDPOINTS         16    /* Max number of endpoints per
                                           node */
#define MCAPI_MAX_MESSAGE_SIZE      0xFFFF /* Max message size */
#define MCAPI_MAX_PACKET_SIZE       0xFFFF /* Max packet size */
#define MCAPI_MAX_CHANNEL_HANDLES   16    /* Max number of channel handles
                                           per endpoint */
#define MCAPI_MAX_REQUESTS          64    /* Max number of request handles
                                           per endpoint */
#define MCAPI_MAX_NO_PRORITIES      8     /* Max number of priorities */
#define MCAPI_TRUE                  1
#define MCAPI_FALSE                 0
#define MCAPI_NULL                  0     /* MCAPI Zero value */
#define MCAPI_PORT_ANY              (-1)  /* Create endpoint using the next
                                           available port */
#define MCAPI_INFINITE              (-1)  /* Wait forever, for timeout */
#define OPEN_SEND                   1     /* For channel opening */
#define OPEN_RECV                   2     /* For channel opening */
#define CLOSE_SEND                  3     /* For channel closing */
#define CLOSE_RECV                  4     /* For channel closing */

/*
 * MCAPI Status codes
 */
enum mcapi_status_codes {
      MCAPI_SUCCESS,           /* Indicates operation was successful */
      MCAPI_INCOMPLETE,        /* Indicates operation has not completed */
      MCAPI_EATTR_INCOMP,      /* Connection of endpoints with incompatible
                                attributes not allowed */
      MCAPI_ECHAN_OPEN,        /* A channel is open, deletion is not allowed
                                */
      MCAPI_ECHAN_TYPE,        /* Attempt to open a packet channel on an
                                endpoint that has been connected with a
                                different channel type */
      MCAPI_ECONNECTED,        /* A channel connection has already been
                                established for one or both of the specified
                                endpoints */
      MCAPI_ENOT_CONNECTED,    /* The channel is not connected (cannot be
                                opened) */
      MCAPI_ENOT_OPEN,         /* The channel is not open (cannot be closed)
                                */
      MCAPI_EDIR,              /* Attempt to open a send handle on a port
                                that was connected as a receiver, or vice
                                versa */
      MCAPI_EEP_NOTALLOWED,    /* Endpoints cannot be created on this node */
      MCAPI_EMESS_LIMIT,       /* The message size exceeds the maximum size
                                allowed by the MCAPI implementation */
      MCAPI_ENO_BUFFER,        /* No more message/packet buffers available */
      MCAPI_ENO_INIT,          /* The MCAPI environment could not be
                                initialized */
      MCAPI_ENODE_NOTINIT,     /* The MCAPI environment is not initialized */
      MCAPI_ENO_FINAL,         /* The MCAPI environment could not be
                                finalized */
      MCAPI_ENO_MEM,           /* Requested size exceeds available memory */
      MCAPI_ENO_REQUEST,       /* No more request handles available */
      MCAPI_ENODE_NOTVALID,    /* The parameter is not a valid node */
      MCAPI_ENOT_ENDP,         /* Argument is not a valid endpoint descriptor
                                */
      MCAPI_ENOT_OWNER,        /* An endpoint can only be deleted by its
                                creator */
```

```
      MCAPI_ENOT_HANDLE,      /* Argument is not a channel handle */
      MCAPI_ENOTREQ_HANDLE,   /* Argument is not a valid request handle */
      MCAPI_EPACK_LIMIT,      /* The message size exceeds the maximum size
                              allowed by the MCAPI implementation */
      MCAPI_EPARAM,           /* Unknown attribute number or incorrect
                              attribute size */
      MCAPI_EPORT_NOTVALID,   /* The parameter is not a valid port */
      MCAPI_EREQ_CANCELED,    /* The request was already canceled */
      MCAPI_EPRIO,            /* Incorrect priority level */
      MCAPI_ETRUNCATED,       /* The message size exceeds the buffer size */
      MCAPI_ENOT_VALID_BUF,   /* Not a valid buffer descriptor */
      MCAPI_ESCL_SIZE,        /* Incorrect scalar size */
      MCAPI_EREQ_TIMEOUT,     /* The operation timed out */
      MCAPI_EENDP_LIMIT,      /* Exceeded maximum number of endpoints */
      MCAPI_INITIALIZED,      /* MCAPI has been initialized */
      MCAPI_EREAD_ONLY,       /* Attribute change not allowed */
      MCAPI_EPARAM_ERROR,     /* Incorrect parameter */
      MCAPI_EENDP_ISCREATED,  /* The endpoint is already created */
      MCAPI_ERROR             /* Used to re-set the status code to a non-
                              zero value */
};

/*
 * MCAPI Status Flags
 */
#define MCAPI_CREATED     0x00000001  /* The endpoint is created */
#define MCAPI_CONNECTED   0x00000002  /* The endpoint is connected */
#define MCAPI_OPEN        0x00000010  /* The channel is open */
#define MCAPI_PKT         0x00000020  /* The channel is a packet channel */
#define MCAPI_SCL         0x00000040  /* The channel is a scalar channel */
#define MCAPI_SEND        0x00000080  /* Enpoint is the send side of the
                                         channel */
#define MCAPI_RECEIVE     0x00000100  /* Enpoint is the receive side of the
                                         channel */
#define MCAPI_GET_PENDING 0x00000200  /* One or more get enpoint are pending
                                         */

/*
 * Non-blocking commands
 */
enum mcapi_nonblock_commands {
      ENDPOINT_GET,           /* Get endpoint */
      MESSAGE_SEND,           /* Message send */
      MESSAGE_RECEIVE,        /* Message receive */
      PKTCHAN_CONNECT,        /* Packet channel connect */
      PKTCHAN_OPEN_RECV,      /* Packet channel open receive side */
      PKTCHAN_OPEN_SEND,      /* Packet channel open send side */
      PKTCHAN_SEND,           /* Packet channel send */
      PKTCHAN_RECV,           /* Packet channel receive */
      PKTCHAN_CLOSE_RECV,     /* Packet channel close receive side */
      PKTCHAN_CLOSE_SEND,     /* Packet channel send receive side */
      SCLCHAN_CONNECT,        /* Scalar channel connect */
      SCLCHAN_OPEN_RECV,      /* Scalar channel open receive side */
      SCLCHAN_OPEN_SEND,      /* Scalar channel open send side */
      SCLCHAN_CLOSE_RECV,     /* Scalar channel close receive side */
      SCLCHAN_CLOSE_SEND      /* Scalar channel close send side */
};

/*
 * Endpoint attribute numbers
 */
```

```
enum mcapi_attrubute_numbers {
      MCAPI_ATTR_NO_PRIORITIES,           /* Number of priorities */
      MCAPI_ATTR_NO_BUFFERS,              /* Number of priorities */
      MCAPI_ATTR_BUFFER_SIZE,             /* Buffer size */
      MCAPI_ATTR_BUFFER_TYPE,             /* Buffer type, FIFO */
      MCAPI_ATTR_MEMORY_TYPE,             /* Shared/local (0-copy) */
      MCAPI_ATTR_TIMEOUT,                 /* Timeout */
      MCAPI_ATTR_ENDP_PRIO,               /* Priority on connected endpoint
                                          */
      MCAPI_ATTR_ENDP_STATUS,             /* Endpoint status, connected,
                                          open etc. */
      MCAPI_ATTR_RECV_BUFFERS_AVAILABLE   /* Available receive buffers */
};

/* MCAPI Attributes */
/* Implementations may desigante some or all attributes as read-only */

/* MCAPI_ATTR_NO_PRIORITIES */
typedef mcapi_int_t mcapi_attr_no_priorities;

/* MCAPI_ATTR_NO_BUFFERS */
typedef mcapi_int_t mcapi_attr_no_buffers;

/* MCAPI_ATTR_BUFFER_SIZE */
typedef mcapi_int_t mcapi_attr_message_buffer_size;

/* MCAPI_ATTR_BUFFER_TYPE */
typedef mcapi_int_t mcapi_attr_buffer_type;

/* MCAPI_ATTR_MEMORY_TYPE */
typedef mcapi_int_t mcapi_attr_memory_type;

/* MCAPI_ATTR_TIMEOUT */
typedef mcapi_timeout_t mcapi_attr_timeout;

/* MCAPI_ATTR_ENDP_PRIO */
typedef mcapi_uint_t mcapi_attr_endp_priority;

/* MCAPI_ATTR_ENDP_STATUS */
typedef mcapi_uint_t mcapi_attr_endp_status;

/* MCAPI_ATTR_RECV_BUFFERS_AVAILABLE */
typedef mcapi_uint_t mcapi_attr_recv_buffers_available;

/* In/out parameter indication macros */
#ifndef MCAPI_IN
#define MCAPI_IN const
#endif /* MCAPI_IN */

#ifndef MCAPI_OUT
#define MCAPI_OUT
#endif /* MCAPI_OUT */


/* Alignment macros */
#ifdef __GNUC__
#define MCAPI_DECL_ALIGNED __attribute__ ((aligned (32)))
#else
#ifndef WIN32
#error /* "MCAPI_DECL_ALIGNED alignment macro currently only suports GNU
compiler" */
```

```
#endif /* WIN32 */
#endif /* __GNUC__ */

#define MCAPI_BUF_ALIGN

#ifndef LIB_BUILD


/*
 * Function prototypes
 */


/* Initialization, node and endpoint management */

extern void mcapi_initialize(
      MCAPI_IN mcapi_node_t node_id,
      MCAPI_OUT mcapi_version_t* mcapi_version,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_finalize(
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern mcapi_uint_t mcapi_get_node_id(
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern mcapi_endpoint_t mcapi_create_endpoint(
      MCAPI_IN mcapi_port_t port_num,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern mcapi_endpoint_t mcapi_get_endpoint_i(
      MCAPI_IN mcapi_node_t node_id,
      MCAPI_IN mcapi_port_t port_id,
      MCAPI_OUT mcapi_endpoint_t* endpoint,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_delete_endpoint(
      MCAPI_IN mcapi_endpoint_t endpoint,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_get_endpoint_attribute(
      MCAPI_IN mcapi_endpoint_t endpoint,
      MCAPI_IN mcapi_uint_t attribute_num,
      MCAPI_OUT void* attribute,
      MCAPI_IN size_t attribute_size,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_set_endpoint_attribute(
      MCAPI_IN mcapi_endpoint_t endpoint,
      MCAPI_IN mcapi_uint_t attribute_num,
      MCAPI_OUT const void* attribute,
      MCAPI_IN size_t attribute_size,
      MCAPI_OUT mcapi_status_t* mcapi_status
);


/* Message functions */

extern void mcapi_msg_send_i(
      MCAPI_IN mcapi_endpoint_t send_endpoint,
```

```
      MCAPI_IN mcapi_endpoint_t receive_endpoint,
      MCAPI_IN void* buffer,
      MCAPI_IN size_t buffer_size,
      MCAPI_IN mcapi_priority_t priority,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_msg_send(
      MCAPI_IN mcapi_endpoint_t send_endpoint,
      MCAPI_IN mcapi_endpoint_t receive_endpoint,
      MCAPI_IN void* buffer,
      MCAPI_IN size_t buffer_size,
      MCAPI_OUT mcapi_priority_t priority,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_msg_recv_i(
      MCAPI_IN mcapi_endpoint_t receive_endpoint,
      MCAPI_OUT void* buffer,
      MCAPI_IN size_t buffer_size,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_msg_recv(
      MCAPI_IN mcapi_endpoint_t receive_endpoint,
      MCAPI_OUT void* buffer,
      MCAPI_IN size_t buffer_size,
      MCAPI_OUT size_t* received_size,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern mcapi_uint_t mcapi_msg_available(
      MCAPI_IN mcapi_endpoint_t receive_endoint,
      MCAPI_OUT mcapi_status_t* mcapi_status
);


/* Packet channel functions */

extern void mcapi_connect_pktchan_i(
      MCAPI_IN mcapi_endpoint_t send_endpoint,
      MCAPI_IN mcapi_endpoint_t receive_endpoint,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_open_pktchan_recv_i(
      MCAPI_OUT mcapi_pktchan_recv_hndl_t* recv_handle,
      MCAPI_IN mcapi_endpoint_t receive_endpoint,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_open_pktchan_send_i(
      MCAPI_OUT mcapi_pktchan_send_hndl_t* send_handle,
      MCAPI_IN mcapi_endpoint_t send_endpoint,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_pktchan_send_i(
      MCAPI_IN mcapi_pktchan_send_hndl_t send_handle,
      MCAPI_IN void* buffer,
      MCAPI_IN size_t size,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
```

```
);
extern void mcapi_pktchan_send(
      MCAPI_IN mcapi_pktchan_send_hndl_t send_handle,
      MCAPI_IN void* buffer,
      MCAPI_IN size_t size,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_pktchan_recv_i(
      MCAPI_IN mcapi_pktchan_recv_hndl_t receive_handle,
      MCAPI_OUT void** buffer,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_pktchan_recv(
      MCAPI_IN mcapi_pktchan_recv_hndl_t receive_handle,
      MCAPI_OUT void** buffer,
      MCAPI_OUT size_t* received_size,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern mcapi_uint_t mcapi_pktchan_available(
      MCAPI_IN mcapi_pktchan_recv_hndl_t receive_handle,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_pktchan_free(
      MCAPI_IN void* buffer,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_pktchan_recv_close_i(
      MCAPI_IN mcapi_pktchan_recv_hndl_t receive_handle,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_pktchan_send_close_i(
      MCAPI_IN mcapi_pktchan_send_hndl_t send_handle,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);


/* Scalar channel functions */

extern void mcapi_connect_sclchan_i(
      MCAPI_IN mcapi_endpoint_t send_endpoint,
      MCAPI_IN mcapi_endpoint_t receive_endpoint,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_open_sclchan_recv_i(
      MCAPI_OUT mcapi_sclchan_recv_hndl_t* receive_handle,
      MCAPI_IN mcapi_endpoint_t receive_endpoint,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_open_sclchan_send_i(
      MCAPI_OUT mcapi_sclchan_send_hndl_t* send_handle,
      MCAPI_IN mcapi_endpoint_t  send_endpoint,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_sclchan_send_uint64(
      MCAPI_IN mcapi_sclchan_send_hndl_t send_handle,
```

```
      MCAPI_IN mcapi_uint64_t dataword,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_sclchan_send_uint32(
      MCAPI_IN mcapi_sclchan_send_hndl_t send_handle,
      MCAPI_IN mcapi_uint32_t dataword,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_sclchan_send_uint16(
      MCAPI_IN mcapi_sclchan_send_hndl_t send_handle,
      MCAPI_IN mcapi_uint16_t dataword,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_sclchan_send_uint8(
      MCAPI_IN mcapi_sclchan_send_hndl_t send_handle,
      MCAPI_IN mcapi_uint8_t dataword,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern mcapi_uint64_t mcapi_sclchan_recv_uint64(
      MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern mcapi_uint32_t mcapi_sclchan_recv_uint32(
      MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern mcapi_uint16_t mcapi_sclchan_recv_uint16(
      MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern mcapi_uint8_t mcapi_sclchan_recv_uint8(
      MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern mcapi_uint_t mcapi_sclchan_available(
      MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_sclchan_recv_close_i(
      MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_sclchan_send_close_i(
      MCAPI_IN mcapi_sclchan_send_hndl_t send_handle,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);


/* Non-blocking management functions */

extern mcapi_boolean_t mcapi_test(
      MCAPI_IN mcapi_request_t* request,
      MCAPI_OUT size_t* size,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern mcapi_boolean_t mcapi_wait(
      MCAPI_IN mcapi_request_t* request,
      MCAPI_OUT size_t* size,
      MCAPI_OUT mcapi_status_t* mcapi_status
```

```
);
extern int mcapi_wait_any(
      MCAPI_IN size_t number,
      MCAPI_IN mcapi_request_t** requests,
      MCAPI_OUT size_t* size,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
extern void mcapi_cancel(
      MCAPI_IN mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);


#endif /* LIB_BUILD */

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* MCAPI_H */
```

6 FAQ

Q: Is a reference implementation available? What is the intended purpose of the reference implementation?

A: A reference implementation is planned in the future. The current plan is to receive feedback on the draft specification and make modifications based upon the feedback. When the specification is finalized, the MCAPI working group will announce the plans and schedule for such an implementation. The reference implementation models the functionality of the specification and does not intend to be a high performance implementation.

Q: Does MCAPI provide binary interoperability?

A: The initial version of MCAPI provides a set of communications primitives that supports software source code portability. Interoperability is not addressed at this stage and is left to the respective MCAPI implementations. Implementers should however be encouraged to ensure implementation interoperability, as much as possible. We may consider an interoperable messaging protocol in future MCAPI versions.

Q: What is the rationale behind MCAPI's use of unidirectional channels as opposed to bidirectional channels?

A: MCAPI is meant to serve as an extremely lightweight and thin communications API for use by applications and by other messaging systems (such as a light-weight sockets layer or a zero copy transfer layer) layered on top of it. MCAPI is expected to run on both multicore processors running real operating systems and running extremely thin run-time environments in bare-metal mode. MCAPI is also meant largely for on-chip multicore environments where communications is reliable.

MCAPI chose to go with unidirectional channels because they use less resources on both the send and receive sides. A bidirectional channel would need to consume twice the resources on both the send and receive sides (for example, hardware queues).

Many of our use cases require only unidirectional channels. Although unidirectional channels could be implemented using a bidirectional channel, it would waste half the resources. Conversely, bidirectional channels can be implemented on top of unidirectional channels when desired. Locking mechanisms on the send and receive sides provided by the underlying system on top of which MCAPI is built (or provided by a layer such as RAPI) can be further used to achieve atomicity between sends and receives on each side.

Bidirectional channels are particularly effective in error-prone environments. MCAPI's target is multicore processors, where the communications is expected to be reliable.

Q: How does MCAPI compare to other existing communication protocols for multicore?

A: MCAPI is a low level communication protocol specifically targeting statically configured heterogenous multicore processors. The static property allows implementations to have lower overhead than other communication protocols such a sockets or MPI.

Q: Regarding message sends, when can the sender reuse the buffer?

A: With blocking calls, the send can reuse once the call has returned. With non-blocking calls, it is the responsibility of the sender to verify using API calls that the buffer can be reused.

Q: Can MCAPI support a software-controlled implementation of global power management? If so, how?

A: MCAPI provides communication functionality between different cores in a multicore multiprocessor and can function at user level and system level. If the global power management scheme requires data

to be passed between different cores, an MCAPI implementation could be used to provide the communications functionality.

Q: Will MCAPI require modification of existing operating systems?

A: MCAPI is a communication API and does not define requirements for implementation. It is possible to implement MCAPI on top of a commercial off-the-shelf OS with no OS modifications, however an efficient implementation of MCAPI may require changes to the operating system.

Q: Can MCAPI be implemented in hardware?

A: The goal of MCAPI is to make possible efficient implementation in hardware.

Q: If MCAPI is oriented towards static connection topologies, why support methods like mcapi_delete_endpoint()?

A: The ability to release any resources allocated by an endpoint may be particularly important in hardware implementations where resources are strictly limited. Supporting the deletion of endpoints allows the application programmer to dispose of endpoints that are only necessary for a portion of the application's life.

Q: What facilities are provided for debugging MCAPI programs?

A: The MCAPI API is designed to be implemented as a C library, using standard tools and enabling the use of standard debuggers and profilers. The MCAPI error codes should also help give some visibility into what sort of program errors have occurred. The MCAPI channel-connection mechanism is designed to work with static layout tools, so that a channel topology can be defined and debugged outside of the program itself. We also expect that some implementations will provide tools for dynamically collecting information like which channels have been connected and how many packets have been transferred.

Q: Why doesn't the MCAPI API support one to multiple, multiple to one and barrier type communications?

A: To keep the MCAPI API simple and allow for efficient implementations more complex functions that can be layered on top of MCAPI are not part of the MCAPI specification.

Q: Why doesn't MCAPI provide name service functionality?

A: The MCAPI API is meant for multicore chips or multi-chip boards which are static environments in the sense that the number of cores is always the same. A naming service is therefore not needed and would add unnecessary complexity. A naming service could be implemented on top of MCAPI.

Q: What happens if my cores have different byte order (endian-ness)?

A: The MCAPI API specification doesn't state how to implement an MCAPI API. The specific implementation would have to address different byte order.

Q: How do I use my MCAPI calls so that the source code is portable between implementations with different node numbering policies?

A: To make code portable between implementations that allow only one node number (and mcapi_init call) per process, and those that allow one node number (and mcapi_init call) per thread, the code should only make one mcapi_init call per process. If there are multiple threads within the process, they should use different port numbers
(endpoints) for communication.

Similarly, to make code portable between implementations that allow only one node number (and mcapi_init call) per processor, and those that allow one node number (and mcapi_init call) per task, the code should only make one mcapi_init call per processor. If there are multiple tasks running on the processor, they should use different port numbers (endpoints) for communication.

Q: Does the MCAPI spec provide for multiple readers or multiple writers on an endpoint?

A: No the spec does not specifically provide for this. However, it does not prevent it either. An MCAPI implementation can choose to provide a mechanism whereby multiple endpoints resolve to essentially the same destination. For example, if an SoC contained a hardware device which managed multiple hardware queues used for moving messages around the SoC, an MCAPI implementation could allow multiple writers to a single hardware queue by simply mapping a set of distinct send endpoints to that single hardware queue. The implementation might choose to do this by assigning a range of port_ids to map to the single hardware device.

For example, assume the MCAPI implementation identifies the hardware device that provided the hardware managed queues as node_id 10, and the range of ports 0..16 is assigned to map to hardware queue zero within that device.

Then a send to endpoint <10,0> as well as a send to endpoint <10,1> would both map to a send to queue zero in the hardware device. Different nodes within the system wishing to send to the queue would have to allocate a unique send endpoint, but the sends would all end up in queue zero in the hardware accelerator. Similarly, an MCAPI implementation could map multiple receive endpoints to a single hardware queue. In this example any reads from multiple endpoints mapped to the single hardware queue would be serviced on a first come, first served basis. This would be suitable for a load balancing application, but would not be sufficient to serve as a multicast operation. Support for multicast operations will be considered in future MCAPI specification releases.

Q: How can I implement callback capability with MCAPI?

A: Although MCAPI does not directly provide a callback capability, MCAPI is

designed to have many kinds of application services layered on top of it, and callbacks can be done this way. This approach would require you to use the MCAPI non-blocking send or receive functions, and to write a new function which would take the mcapi_request_t returned from calling these MCAPI functions along with a pointer to your callback function as parameters. This new function could use mcapi_test, mcapi_wait, or mcapi_wait_any to determine when the outstanding MCAPI request has completed and then call your callback function. This might be implemented by having a separate thread monitoring a list of outstanding requests and providing callbacks on completion. If you cannot create multithreaded applications your callback solution may be more difficult to implement.

Q: I'd like to have a name service for looking up MCAPI endpoints. How can I achieve this functionality?

The MCAPI standard defines a communication layer that allows easy creation of a name server. The application can choose a statically determined (node, port) pair where the name service will run. For example, node 0, port 0 might be a good choice. The name server process can be started by calling mcapi_initialize(0) to bind itself to node 0 and then calling mcapi_create_endpoint(0, &status). The name server can then use message send and receive operations to implement its service. Clients can discover the name server by invoking mcapi_get_endpoint(0, 0, &status) to get the its endpoint_t. Given the endpoint_t, each client can send and receive messages as required by your name service protocol.

7 Use Cases

NOTE: The use case examples will be updated in a future revision, once MCAPI implementations are available.

7.1 Example Usage of Static Naming for Initialization

MCAPI's static tuple based naming mechanism makes it straightforward to implement a simple initialization scheme, including third party set up of static connections. This example describes how a packet channel can be created and used using the static tuple based naming scheme.

```
#define SENDER_NODE 0
#define SEND_PORT_ID 17

#define RECEIVER_NODE 1
#define RECV_PORT_ID 37
```

Sender Process:

```
endpoint_t send_endpoint = mcapi_create_endpoint(SEND_PORT_ID);
```

Receiver Process:

```
endpoint_t receive_endpoint = mcapi_create_endpoint(RECV_PORT_ID);
```

The connection can now be established by the receiver, the sender, or a third party process. We will use the example of a third party process.

Third party process:

```
endpoint_t send_endpoint = mcapi_get_endpoint(SENDER_NODE, SEND_PORT_ID);
endpoint_t receive_endpoint = mcapi_get_endpoint(RECEIVER_NODE,
                                                 RECV_PORT_ID);
mcapi_connect_pktchan(send_endpoint, receive_endpoint, …)
```

Thus, the performance impact of a global name-service lookup is entirely in the mcapi_get_endpoint() call.

7.2 Example Initialization (Discovery and Bootstrapping) of Dynamic Endpoints

This example describes how MCAPI performs discovery and bootstrapping when the static naming based on tuples is not being used, rather the dynamic endpoint scheme is used.

Whether communication is by messages or by channels, MCAPI requires a sending node to have available an endpoint on the receiving node in order to communicate. Thus there is the following discovery and bootstrapping issue in MCAPI when static naming is not in use (and in any dynamic communication API for that matter): How to transfer the first endpoint from a receiver to a sender? This

section proposes a discovery model, which addresses the issue of how an MCAPI sender can bootstrap itself by finding the endpoint for a receiver, essentially before any user-level communication has been established.

The basic idea is to allow the MCAPI runtime system to facilitate the creation of a root endpoint that is visible to all the nodes in the system. An endpoint can “publish” itself to the communications layer as the “root” or first-level name server of the namespace using a statically known node number. For example, the system could have exactly one statically numbered node, node 0. All other nodes in the system would then register with node 0 via asynchronous messaging, sending it messages to let it know that they exist and what endpoints they have dynamically created. Similarly, dynamic nodes and endpoints can discover each other by sending messages to the 'root node'. The format of the message is user defined.

7.3 Automotive Use Case

7.3.1 Characteristics

7.3.1.1 Sensors

Tens to hundreds of sensor inputs read on periodic basis. Each sensor is read and its data are processed by a scheduled task.

7.3.1.2 Control Task

A control task takes sensor values and computes values to apply to various actuators in the engine.

7.3.1.3 Lost Data

Lost data is not desirable, but old data quickly becomes irrelevant, most recent sample is most important.

7.3.1.4 Types of Tasks

Consists of both control and signal processing, especially FFT.

7.3.1.5 Load Balance

The load balance changes as engine speed increases. The frequency at which the control task must be run is determined by the RPM of the engine.

7.3.1.6 Message Size and Frequency

Messages are expected to be small and message frequency is high.

7.3.1.7 Synchronization

Synchronization between control and data tasks should be minimal to avoid negative impacts on latency of the control task; if shared memory is used it will always be one task writing and the other reading, so synch is not required or desirable; deadlock will not occur but old data may be used if an update is not ready.

7.3.2 Key functionality requirements

7.3.2.1 Control Task

There must be a control task collecting all data and calculating updates; this task must update engine parameters continuously; Updates to engine parameters must occur when the engine crankshaft is at a particular angle, so the faster the engine is running, the more frequently this task must run.

7.3.2.2 Angle Task

There must be a data task to monitor engine RPM and schedule the control task.

7.3.2.3 Data Tasks

There must be a set of tens to hundreds of task to poll sensors; the task must communicate this data to the control task.

7.3.3 Context/Constraints

7.3.3.1 Operating System

Often there is no commercial operating system involved, although the notion of time critical tasks and task scheduling must be supported by some type of executive; however this may be changing; possible candidates are OSEK, or other RTOS.

7.3.3.2 Polling/Interrupts

Sensor inputs may be polled and/or associated with interrupts.

7.3.3.3 Reliability

Sensors assumed to be reliable; interconnect assumed to be reliable; task completion within scheduled deadline is assumed to be reliable for the control task, and less reliable for the data tasks.

7.3.4 Metrics

7.3.4.1 Latency of the control task

Dependent on engine RPM. At 1800 RPM the task must complete every 33.33ms, and at 9000 RPM the task must complete every 6.667ms.

7.3.4.2 # dropped sensor readings

Ideally zero.

7.3.4.3 Latencies of data tasks

Ideally the sum of the latencies plus message send/receive times should be < latency of the control loop given current engine RPM. In general, individual tasks are expected to complete in times varying from 1ms up to 1600ms, depending on the nature of the sensor and the type of processing required for its data.

7.3.4.4 Code size

Automotive customers expect their code to fit into on chip SRAM. Current generation of chips often has 1Mb of SRAM, with 2Mb on the near horizon.

7.3.5 Possible Factorings

- 1 gp core for control, 1 gp core for data
- 1 gp core for control/data, dedicated SIMD core for signal processing, other SP cores for remainder of data processing
- 1 core per cylinder, or 1 core per group of cylinders

7.3.6 MCAPI Requirements Implications

- The control task must be able to determine if data is available from each data task and if not the task should be able to proceed in a non-blocking fashion using the last data from the sensor in question; this implies some form of non-blocking msg test or select mechanism [Automotive:7.3.2.1]

- It would be desirable for the control task to use MCAPI methods to send updates to actuators as ‘messages’, this implies some form of underlying efficient driver should be implemented; the send should be non-blocking [Automotive:7.3.2.1]
- It would be desirable for the task to read data from the sensor using MCAPI methods, this implies some sort of driver implementation underneath the MCAPI [Automotive:7.3.2.2]
- The data task should be able to do a non-blocking message send to the control task [Automotive:7.3.2.2]

7.3.7 Mental Models

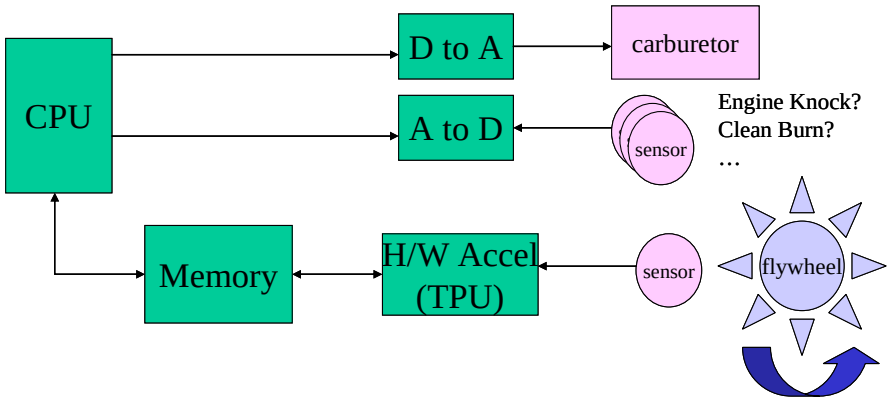


Figure 1 – example hardware

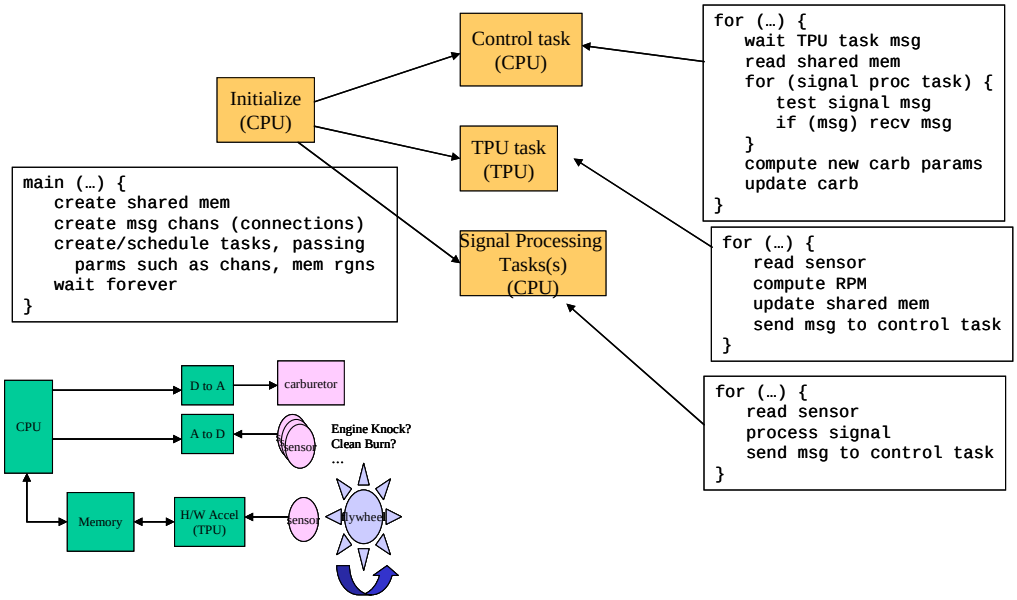


Figure 2 – A possible mapping

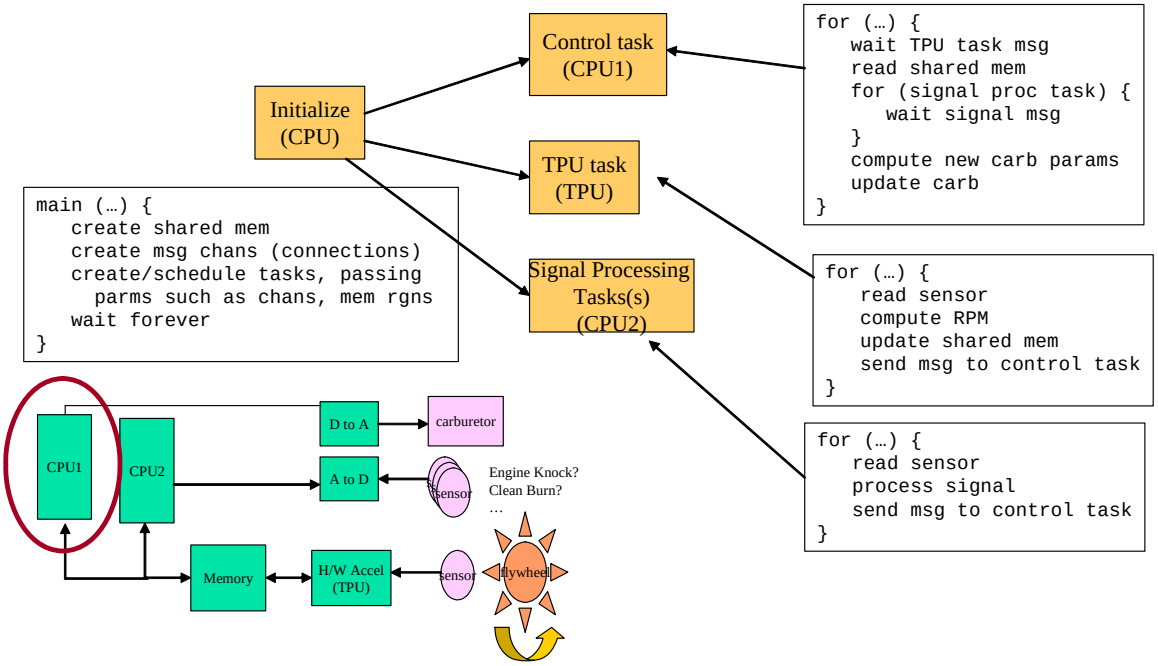


Figure 3 – Alternative Hardware

7.3.8 Applying the API draft to the pseudo-code

7.3.8.1 Initial Mapping

```
//////////////////////////////////////
// The control task
//////////////////////////////////////
void Control_Task(void) {
   char* sMem;
   uint8_t tFlag;
   mcapi_endpoint_t tpu_endpt, tpu_remote_endpt;
   mcapi_endpoint_t sig_endpt, sig_remote_endpt;
   mcapi_endpoint_t tmp_endpt;
   mcapi_sclchan_recv_hndl_t tpu_chan;
   mcapi_pktchan_recv_hndl_t sig_chan;
   struct SIG_DATA sDat;
   size_t tSize;
   mcapi_request_t r1, r2;
   mcapi_status_t err;

   // init the system
   mcapi_initialize(CNTRL_NODE, &err);
    CHECK_STATUS(err);

   // first create two local endpoints
   tpu_endpt = mcapi_create_endpoint(CNTRL_PORT_TPU, &err);
    CHECK_STATUS(err);
```

```
sig_endpt = mcapi_create_endpoint(CNTRL_PORT_SIG, &err);
 CHECK_STATUS(err);

// now we get two remote endpoints
mcapi_get_endpoint_i(TPU_NODE, TPU_PORT_CNTRL,
                     &tpu_remote_endpt, &r1, &err);
 CHECK_STATUS(err);

mcapi_get_endpoint(SIG_NODE, SIG_PORT_CNTRL,
                   &sig_remote_endpt, &r2, &err);
 CHECK_STATUS(err);

// wait on the endpoints
while (!((mcapi_test(&r1,NULL,&err)) &&
         (mcapi_test(&r2,NULL,&err))) {
    // KEEP WAITING
}

// allocate shared memory and send the ptr to TPU task
sMem = shmemget(32);

tmp_endpt = mcapi_create_anonymous_endpoint(&err);
CHECK_STATUS(err);

mcapi_msg_send(tmp_endpt, tpu_remote_endpt, sMem,
               sizeof(sMem), &err);
CHECK_STATUS(err);

// connect the channels
mcapi_connect_sclchan_i(tpu_endpt, tpu_remote_endpt,
                        &r1, &err);
 CHECK_STATUS(err);

mcapi_connect_pktchan_i(sig_endpt, sig_remote_endpt,
                        &r2, &err);
 CHECK_STATUS(err);

// wait on the connections
while (!((mcapi_test(&r1,NULL,&err)) &&
         (mcapi_test(&r2,NULL,&err))) {
    // KEEP WAITING
}

// now open the channels
mcapi_open_sclchan_recv_i(&tpu_chan, tpu_endpt,
                          &r1, &err);
 CHECK_STATUS(err);

mcapi_open_pktchan_recv_i(&sig_chan, sig_endpt,
                          &r2, &err);
 CHECK_STATUS(err);
```

```
   // wait on the channels
   while (!((mcapi_test(&r1,NULL,&err)) &&
            (mcapi_test(&r2,NULL,&err))) {
       // KEEP WAITING
   }

   // now ALL of the bootstrapping is finished
   // we move to the processing phase below

   while (1) {
     // wait for TPU to indicate it has updated
     // shared memory, indicated by receipt of a flag
     tFlag = mcapi_sclchan_recv_uint8(tpu_chan, &err);
      CHECK_STATUS(err);

      // read the shared memory
      if (sPtr[0] != 0) {
         // process the shared memory data
      } else {
         // PANIC -- there was an error with the shared mem
     }

     // now get data from the signal processing task
     // would be a loop if there were multiple sig tasks
     mcapi_pktchan_recv(sig_chan, (void **) &sDat,
                        &tSize, &err);
      CHECK_STATUS(err);

      // Compute new carb params & update carb
   }
}
```

```
//////////////////////////////////////
// The TPU task
//////////////////////////////////////
void TPU_Task() {
   char* sMem;
   size_t msgSize;
   mcapi_endpoint_t cntrl_endpt, cntrl_remote_endpt;
   mcapi_sclchan_send_hndl_t cntrl_chan;
   mcapi_request_t r1;
   mcapi_status_t err;

   // init the system
   mcapi_initialize(TPU_NODE, &err);
    CHECK_STATUS(err);

   cntrl_endpt = mcapi_create_endpoint(TPU_PORT_CNTRL, &err);
    CHECK_STATUS(err);

   mcapi_get_endpoint_i(CNTRL_NODE, CNTRL_PORT_TPU,
                        &cntrl_remote_endpoint, &r1, &err);
    CHECK_STATUS(err);

    // wait on the remote endpoint
    mcapi_wait(&r1,NULL,&err);
    CHECK_STATUS(err);

   // now get the shared mem ptr
   mcapi_msg_recv(cntrl_endpt, &sMem, sizeof(sMem), &msgSize, &err);
   CHECK_MEM(sMem);
   CHECK_STATUS(err);

   // NOTE – connection handled by control task
   // open the channel
   mcapi_open_sclchan_send_i(&cntrl_chan, cntrl_endpt, &r1, &err);
    CHECK_STATUS(err);

    // wait on the open
    mcapi_wait(&r1,NULL,&err);
    CHECK_STATUS(err);

    // ALL bootstrapping is finished, begin processing
   while (1) {
      // do something that updates shared mem
      sMem[0] = 1;

      // send a scalar flag to cntrl process
      // indicating sMem has been updated
      mcapi_sclchan_send_uint8(cntrl_chan, (uint8_t) 1,
                               &err);
     CHECK_STATUS(err);
   }
}
```

```
//////////////////////////////////////
// The SIG Processing Task
//////////////////////////////////////
void SIG_task() {
   mcapi_endpoint_t cntrl_endpt, cntrl_remote_endpt;
   mcapi_pktchan_send_hndl_t cntrl_chan;
   mcapi_request_t r1;
   mcapi_status_t err;

   // init the system
   mcapi_initialize(SIG_NODE, &err);
    CHECK_STATUS(err);

   cntrl_endpt =
      mcapi_create_endpoint(SIG_PORT_CNTRL, &err);
    CHECK_STATUS(err);

   mcapi_get_endpoint_i(CNTRL_NODE, CNTRL_PORT_SIG,
                        &cntrl_remote_endpt, &r1, &err);
    CHECK_STATUS(err);

    // wait on the remote endpoint
    mcapi_wait(&r1,NULL,&err);
    CHECK_STATUS(err);

   // NOTE – connection handled by control task
   // open the channel
   mcapi_open_pktchan_send_i(&cntrl_chan, cntrl_endpt,
                             &r1, &err);
    CHECK_STATUS(err);

    // wait on the open
    mcapi_wait(&r1,NULL,&err);
    CHECK_STATUS(err);


   // All boostrap is finished, now begin processing
   while (1) {
        // Read sensor & process signal
        struct SIG_DATA sDat;  // populate this with results

       // send the data to the control process
        mcapi_pktchan_send(cntrl_port, &sDat, sizeof(sDat),
                           &err);
        CHECK_STATUS(err);
    }
}
```

7.3.8.2 Changes required to port to new multicore device

To map this code to additional CPUs, the only change required to this code is in the constant definitions for node and port numbers in the creation of endpoints.

7.3.8.3 Changes required to port to new multicore device

To map this code to additional CPUs, the only change required to this code is in the constant definitions for node and port numbers in the creation of endpoints.

7.4 Multimedia Processing Use Cases

Multimedia processing includes coding and decoding between various audio and video formats. Applications range from low-power mobile devices such as cell phones, with limited resolution and audio quality, to set-top-boxes and HDTV with extremely demanding performance requirements.

The following will review some of the use cases and communications characteristics for the multimedia application domain.

7.4.1 Characteristics

7.4.1.1 Simple Scenario

Figure 1 is a simple illustration of a multicore multimedia architecture. In this scenario, a multicore processor is executing a multimedia application, which is accelerated by a DSP integrated in the multicore device. The application has some data (for example, a video frame encoded in the MPEG4 format), and uses the DSP to decode into an image suitable for display. The data is moved (at least conceptually) to the DSP, the DSP processing runs, and the data is moved back to the general purpose processor (GPP).

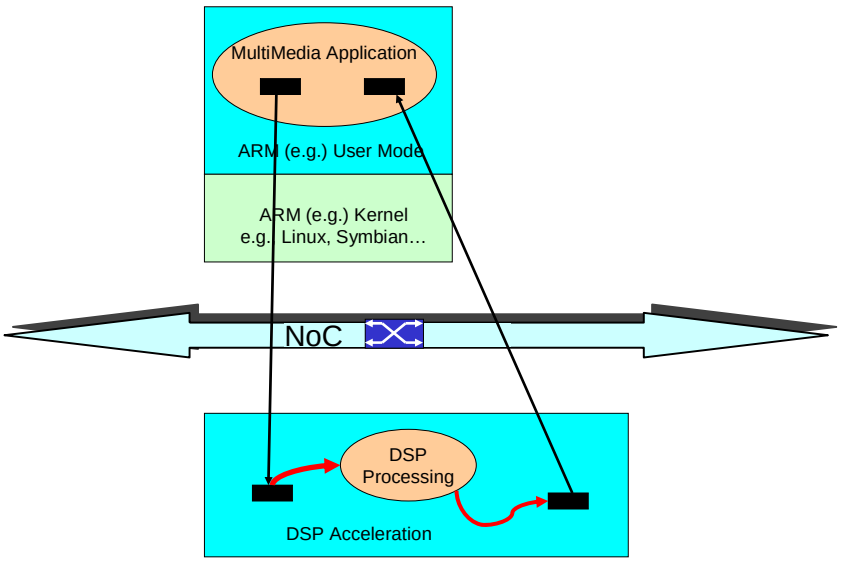


Figure 1 - Simple Multimedia Scenario

Despite the simplicity of the above scenario, a number of characteristics are illustrated:

Characteristic 1 Heterogeneous Processors

The DSP and GPP have different instruction sets and potentially different data representations. The accelerators are typically 16, 24 or 32 bit special purpose devices. The accelerators may have limited code and data spaces. For example, total DSP code space for the application, operating system and communication infrastructure may be under 64K instruction words. In order to save room for applications, the communication infrastructure code footprint is ideally quite small (e.g., less than 1K VLIW instruction words). The communication data footprint is of the same magnitude.

There seems to be a trend towards 32 bit processing and less constrained acceleration devices in the future, although this is not altogether clear, in particular for low-power mobile devices.

In the example, the application is executing in user mode on a general purpose operating system such as Linux. Other operating systems such as QNX Neutrino, VxWorks, WinCE, etc may be used. The DSP is running potentially another standard operating system, or a "home grown" operating system, or perhaps in a "bare metal" environment, with no operating services to speak of.

Characteristic 2 Heterogeneous Communication Infrastructure

Figure 1 shows a network on chip connection between the GPP (in this example an ARM processor) and DSP. This may range from a simple bus to a well-designed network on chip infrastructure. However, (and perhaps more typically today), this could be an ad-hoc collection of buses and bridges, with different DMA engines and other mechanisms for moving data around the system.

Characteristic 3 Heterogeneous Application Environments

Multimedia applications are very complex, and have a number of frameworks such as gstreamer, DirectShow, MDF, OpenMax, etc. The communication infrastructure should "play well" with these environments.

For example, Figure 2 shows the OpenMax multimedia framework from the OpenMax specification[3]. Host-side communication APIs may already be defined by the framework. The MCAPI communication infrastructure is more likely suited to the tunneled communication between acceleration components.

[3] See http://www.khronos.org/openmax/

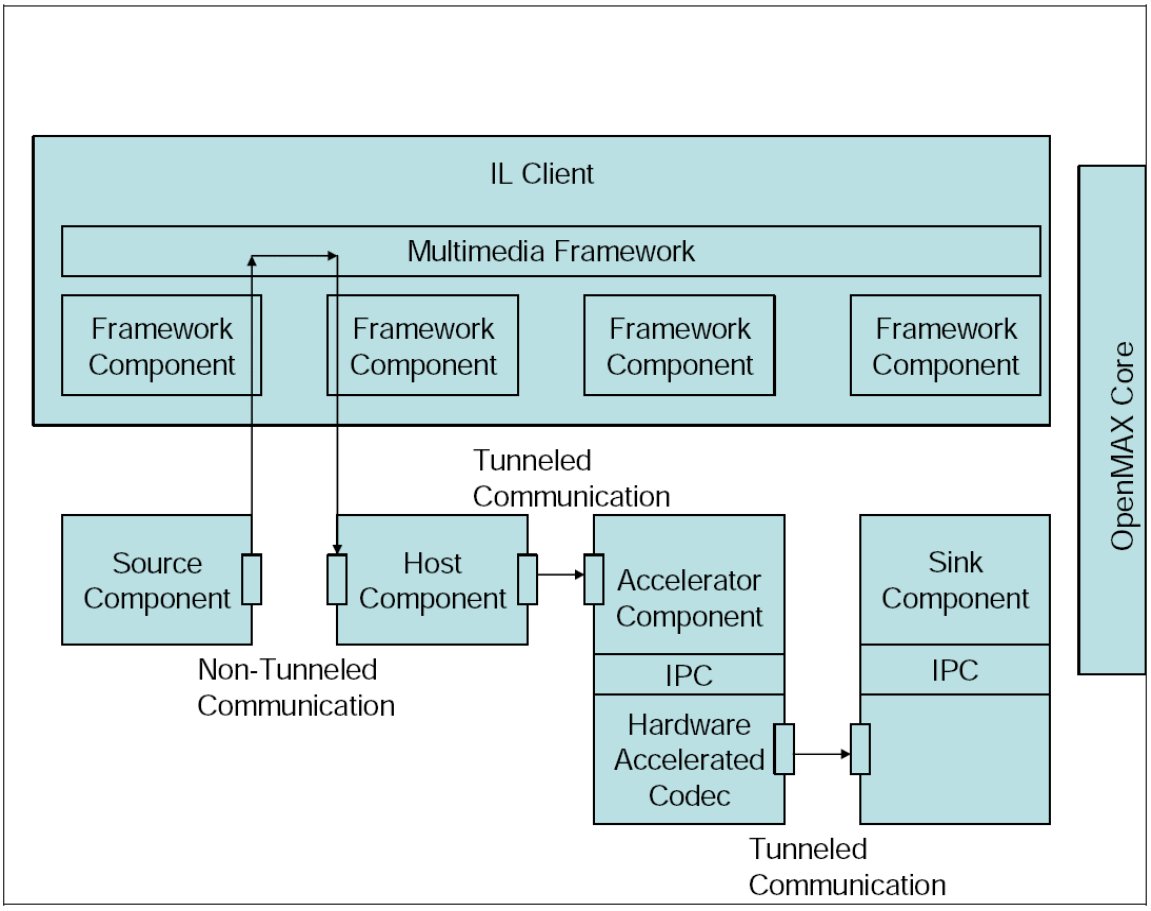


Figure 2 - OpenMax Communication

Characteristic 4 Application Non-Resident Code Size

Due to the large number of standards, encoding formats, processing variants, etc, the non-resident multimedia code size is quite large (millions of lines of code). This is only the code running on the accelerators, and does not include the code on the general purpose processors. Of course, depending on the operating mode, only a small subset of this code will be configured and running at a particular point in time.

Characteristic 5 Client/Server Communication Pattern

In simple configurations as illustrated in Figure 1, the communication follows a client/server pattern. However, as we will see shortly, this is not typically the case.

In the simple case, communication may be considered coarse grained, in that a large chunk of data is passed to the accelerator (for example, a video frame), and the DSP runs for a relatively long period of time to compute the results. Again, this coarse grained behavior is not always the case, as is described below.

Characteristic 6 Coarse Grain Data Rates and Latencies

For the coarse grained scenario, data rates and latencies are relatively conservative. Order of magnitude data sizes are 1K bytes, and rates under 100 Hz.

Characteristic 7 Potential for Zero Copy

Performance and power are key constraints. Unnecessary data movement and/or processing should be avoided where possible. Therefore, although the figure shows distinct data buffers on the GPP and DSP side, in practice the data may reside in a common shared memory. No data is actually copied, when processing control is transferred from the GPP to the DSP (and back again).

However, in many cases, the architecture does not feature a shared memory capability, in which case the data must be moved. Ideally, the communication infrastructure should mask these platform-dependant constraints, and allow zero copy where supportable by the architecture. This permits portable application code.

7.4.1.2 More Complex Scenarios

Figure 3 is a conceptual representation of a more complex scenario. This is motivated by applications such as H.264 encoding/decoding.

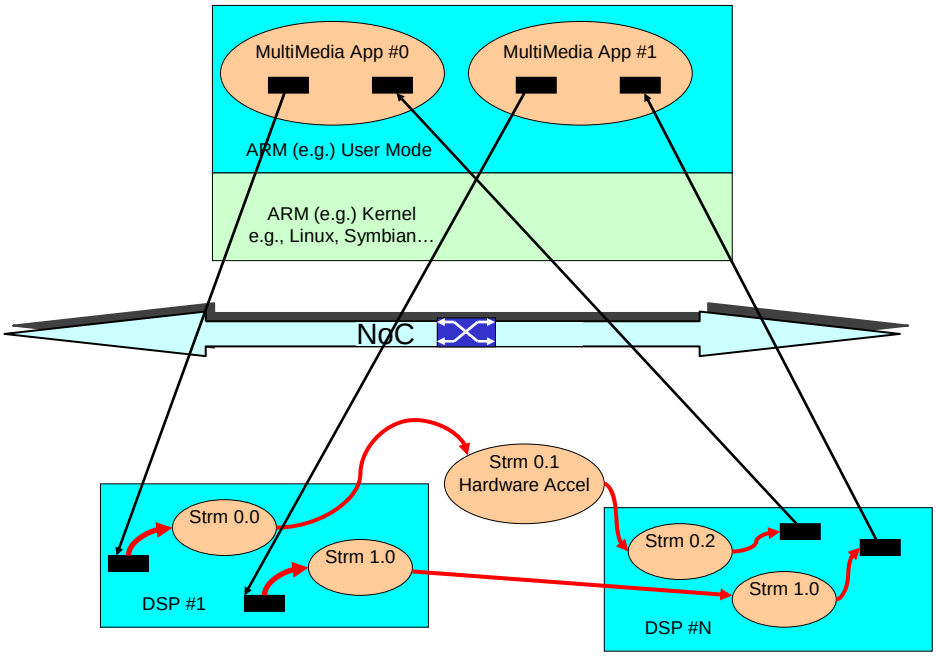


Figure 3 - More Complex Scenarios

Characteristic 8 High Processing Rates

The system computation requirements may exceed 100 GOPS/sec. The following characteristic may be viewed as a corollary:

Characteristic 9 Multi-Stage Processing

Due to extreme compute requirements, no one core can perform all of the processing – processing must be split into several stages. Currently, multimedia devices with order 10 cores are common.

Characteristic 10 Hardware Components

Even when split into multiple stages, computation for a particular stage may exceed the capabilities of a programmable device. In this case, the processing stage may be implemented in hardware. Therefore, the communication infrastructure must support communication to/from hardware devices.

Characteristic 11 Trend to finer-grained processing and communication

Several factors are influencing a trend to finer-grained processing and communication. One factor is the complexity, change, and customization options of the emerging standards. This requires great flexibility in the implementation, which in turn favors an approach which decomposes the problem into many parts, with the option of quickly changing the implementation of the parts. It seems that a natural fallout of this finer-grained decomposition is the requirement for finer-grained communication sizes and rates.

For example: a processor sends order 100 bytes to a hardware block for processing that takes order 100 cycles.

However, many different variants and approaches are in active investigation, the optimal strategy is not clear.

Characteristic 12 Streamed Processing

In the multi-stage configuration, a dataflow or streaming style of computation may be more appropriate. Depending on the granularity of the decomposition, the “tokens” exchanged between stages may be large (e.g., video frames), down to individual data values.

Characteristic 13 Multiple Flows

In many cases, multiple flows must be supported. For example, one or more audio streams, the main video stream, picture-in-picture streams, etc.

Characteristic 14 Dynamic Operation

Multimedia devices are increasingly dynamic. This includes many operating modes (voice calls, digital still camera, video reception and transmission, audio playback and record, etc). Arbitrary subsets of these operating modes may be selected for simultaneous operation by the user at fairly fine time-scales (e.g., in the order of seconds). Within a mode, operation may be dynamic at even finer time scales. For example, packets from a wireless connection may arrive in bursts.

Characteristic 15 Non-deterministic communication rates

In many cases, the amount of data consumed and produced by a processing stage may not be deterministic, resulting in variable communication rates.

Characteristic 16 Low Power Operation

Power is a central concern for mobile devices. Components must be switched off or moved to lower voltage/operating frequency depending on system load. Computation occasionally must be redistributed over the available resources in order to optimize power consumption.

Characteristic 17 Emergent System-Level Properties & Problems

Due to the complexity of the device and the unintended coupling of components, multimedia devices may exhibit difficult to understand “emergent” properties. For example, the system may deadlock, overflow buffers, miss deadlines, or provide “bursty” output, etc. Mechanisms to deal with these problems are very desirable.

Characteristic 18 Quality of Service

Device operation must meet quality of service constraints (processing latency, throughput, jitter, etc). Ideally, the communications infrastructure should support this. Due to high performance requirements, parts of this infrastructure may be in hardware.

Characteristic 19 Security

Multimedia flows have a strong security requirement. It may be required to prevent access to decoded data (for example), even if the operating system(s) have been compromised.

7.4.1.3 Metrics

The following table summarizes some of the quantitative characteristics of the multimedia application domain.

Attribute	Value (ranges)	Notes
Communication code size footprint on accelerators	8 K bytes	Becoming less constrained in the future
Communication data size footprint	8 K bytes	Becoming less constrained in the future
Non-resident code size	millions of lines of code	Increasing in the future
Number of processors	order 10	
Hardware accelerators	under 10	Different accelerators in high-end systems
Communication sizes	1K to under 100 bytes	Trend to finer-grain
Communication latency	10 mSec to under 1 uSec	Trend to finer-grain in future

7.4.2 Key Functionality Requirements

Table 1 summarizes the key requirements of multimedia devices and comments upon the ability MCAPI to meet the requirement.

Table 1 - Multimedia Derived Requirements

Description	Derived From	MCAPI
Low footprint on constrained devices (code and data size)	Characteristic 1	Yes
Framework for heterogeneous data types	Characteristic 1	Yes
Framework for complex transport configuration, quality of service, message delivery semantics and queuing policies (FIFO, most recent only, etc)	Characteristic 2	Some, others can be built on top.
Minimal assumptions about operating environment, and services required	Characteristic 3	Yes
No unnecessary data movement if architecture has supporting features (e.g., shared memory)	Characteristic 7	Yes, zero copy functionality
Allow direct binding to hardware, allowing end-to-end message delivery rates in the order of 100 cycles	Characteristic 8 Characteristic 9 Characteristic 10 Characteristic 11	Yes, no MCAPI feature prevents it
Framework for word-by-word message send/receive	Characteristic 8 Characteristic 9 Characteristic 10 Characteristic 11 Characteristic 12	Yes
Allow blocking and non-blocking operations	Characteristic 3	Yes
Provide message matching framework, to allow selection of messages of particular type, priority, deadline, etc	Characteristic 18	No, can be built on top of MCAPI.
Allow multi-drop messages. E.g., potential for >1 senders to queuing point, >1 readers	Characteristic 16 Characteristic 18	No, can be built on top of MCAPI.

7.4.3 Pseudo-Code Example:

The pseudo-code scenario contains the execution of a DSP algorithm such as a FIR filter on a multicore processor system containing a general purpose processor and DSP. The high level steps are as follows:

1. Initialize communication channels between processors.
2. GPP sends data for processing to DSP.
3. GPP sends code to execute to DSP
4. GPP signals for the DSP to begin execution.
5. GPP retrieves processed data from DSP

```
/* Shared Header File, shared.h */
enum {
  DSP_READY,
  DSP_DATA,
  DSP_CODE,
  DSP_TERMINATE
  DSP_EXECUTE
};


/* General Purpose Processor Code */
#include “shared.h”
```

```
#include <mcapi.h>

/* Predefined Node numbering */
#define GPP_1 0  /* 0-63 reserved for homogenous MC */
#define DSP_1 64 /* 64+ used for DSP nodes */

/* Predefined Port numbering */
#define PORT_COMMAND 0
#define PORT_DATA 1
#define PORT_DATA_REC 2

mcapi_endpoint_t command_endpoint;
mcapi_endpoint_t data_endpoint;
mcapi_endpoint_t remote_command_endpoint;
mcapi_endpoint_t remote_data_endpoint;
mcapi_endpoint_t data_recv_endpoint;
mcapi_endpoint_t remote_data_recv_endpoint ;

mcapi_pktchan_send_hndl_t data_chan;
mcapi_sclchan_send_hndl_t command_chan;
mcapi_pktchan_recv_hndl_t data_recv_chan;

void *data;
int data_size;
void *code;
int code_size;

void initialize_comms()
{
  mcapi_status_t status;

  mcapi_request_t request;

  mcapi_initialize(&status);
  CHECK_STATUS(status);

  command_endpoint = mcapi_create_endpoint(PORT_COMMAND, & status);
  CHECK_STATUS(status);

  data_endpoint = mcapi_create_endpoint(PORT_DATA, & status);
  CHECK_STATUS(status);

  data_recv_endpoint = mcapi_create_endpoint(PORT_DATA_RECV, & status);
  CHECK_STATUS(status);

  remote_command_endpoint = mcapi_get_endpoint(DSP_1, PORT_COMMAND,
                                               &status);
  CHECK_STATUS(status);

  remote_data_endpoint = mcapi_get_endpoint(DSP_1, PORT_DATA, &status);
  CHECK_STATUS(status);

  remote_data_recv_endpoint = mcapi_get_endpoint(DSP_1, PORT_DATA_RECV,
                                                 &status);
  CHECK_STATUS(status);

  mcapi_connect_pktchan(&data_endpoint, remote_data_endpoint, &request,
                        &status);
  CHECK_STATUS(status);

  mcapi_connect_sclchan_i(&command_endpoint, remote_command_endpoint,
```

```
                          &request, &status)
  CHECK_STATUS(status);

  mcapi_connect_pktchan(&data_recv_endpoint, remote_data_recv_endpoint,
                        &request, &status);
  CHECK_STATUS(status);

  mcapi_open_pktchan_send_i(&data_chan, data_endpoint, &request, &status);
  CHECK_STATUS(status);

  mcapi_open_sclchan_send_i(&command_chan, command_endpoint, &request,
                            &status);
  CHECK_STATUS(status);

  mcapi_open_pktchan_recv_i(&data_recv_chan, data_recv_endpoint, &request,
                            &status);
  CHECK_STATUS(status);

}

void send_data(void *data, int size)
{
  mcapi_status_t status;
  mcapi_pktchan_send(data_chan, data, size, &status);
  CHECK_STATUS(status);
}

void send_dsp_cmd(int cmd)
{
  mcapi_status_t status;
  mcapi_sclchan_send_uint32(command_chan, cmd, &status);
  CHECK_STATUS(status);
}

void read_data(void **dst, int *size)
{
  mcapi_status_t status;
  mcapi_pktchan_recv(data_recv_chan, dst, size, &status);
  CHECK_STATUS(status);

}

void shutdown_comms()
{
  mcapi_status_t status;

  mcapi_pktchan_recv_close(data_recv_chan, &status);
  CHECK_STATUS(status);
  mcapi_pktchan_send_close(data_chan, &status);
  CHECK_STATUS(status);
  mcapi_sclchan_send_close(command_chan, &status);
  CHECK_STATUS(status);

  mcapi_delete_endpoint(command_endpoint, &status)
  CHECK_STATUS(status);

  mcapi_delete_endpoint(data_endpoint, &status)
  CHECK_STATUS(status);

  mcapi_delete_endpoint(data_recv_endpoint, &status)
  CHECK_STATUS(status);
```

```
}

void *allocate(int) {
  /* Routine to allocate memory */
}

int perform_multimedia_function(void *code, int code_size, void *data, int
data_size)
{
  initialize_comms();
  send_dsp_cmd(DSP_DATA);
  send_data(data, data_size);
  send_dsp_cmd(DSP_CODE);
  send_data(code, code_size);
  send_dsp_cmd(DSP_EXECUTE);
  read_data(result, size);
  send_dsp_cmd(DSP_TERMINATE);
  shutdown_comms();
}

/* DSP Code */

#include “shared.h”
#include <mcapi.h>


mcapi_endpoint_t dsp_command_endpoint;
mcapi_endpoint_t dsp_data_endpoint;
mcapi_endpoint_t dsp_data_send_endpoint;

mcapi_pktchan_recv_hndl_t data_chan;
mcapi_sclchan_recv_hndl_t command_chan;
mcapi_pktchan_send_hndl_t data_send_chan;

void *buffer;
void *code_buffer;
void *data_buffer;
int code_size;
int data_size;
void *result_buffer;
int result_size;

int dsp_initialize()
{
  mcapi_status_t status;

  mcapi_request_t request;

  mcapi_initialize(&status);
  CHECK_STATUS(status);

  dsp_command_endpoint = mcapi_create_endpoint(PORT_COMMAND, &status);
  CHECK_STATUS(status);

  dsp_data_endpoint = mcapi_create_endpoint(PORT_DATA, &status);
  CHECK_STATUS(status);
  dsp_data_send_endpoint = mcapi_create_endpoint(PORT_DATA_RECV, &status);
  CHECK_STATUS(status);
```

```
  mcapi_open_pktchan_recv_i(&data_chan, dsp_data_endpoint, &request,
                            &status);
  CHECK_STATUS(status);

  mcapi_open_sclchan_recv(&command_chan, dsp_command_endpoint, &request,
                          &status);
  CHECK_STATUS(status);

  mcapi_open_pktchan_send(&data_send_chan, dsp_data_send_endpoint, &request,
                          &status);
  CHECK_STATUS(status);

}

int dsp_shutdown()
{
  mcapi_status_t status;

  mcapi_pktchan_free(data_buffer);

  mcapi_pktchan_free(code_buffer);

  mcapi_pktchan_recv_close(dsp_data_endpoint, &status);
  CHECK_STATUS(status);

  mcapi_sclchan_recv_close(dsp_command_endpoing, &status);
  CHECK_STATUS(status);

  mcapi_pktchan_send_close(dsp_data_send_endpoint, &status);
  CHECK_STATUS(status);

  mcapi_delete_endpoint(command_endpoint, &status);
  CHECK_STATUS(status);

  mcapi_delete_endpoint(data_endpoint, &status);
  CHECK_STATUS(status);

  mcapi_delete_endpoint(dsp_data_endpoint, &status);
  CHECK_STATUS(status);


}

int receive_dsp_cmd()
{
  mcapi_status_t status;
  cmd = mcapi_sclchan_recv_uint32(command_chan, &status);
  CHECK_STATUS(status);
}

int dsp_command_loop()
{
  int command = DSP_READY;
  mcapi_status_t status;

  dsp_initialize();

  while (command != DSP_TERMINATE) {
    switch (command) {
      case DSP_DATA:
        mcapi_pktchan_recv(data_chan, &data_buffer, &data_size, &status);
```

```
        CHECK_STATUS(status);
      break;
      case DSP_CODE:
        mcapi_pktchan_recv(data_chan, &code_buffer, &code_size, &status);
        CHECK_STATUS(status);
        /* Copy code to from buffer to local execute memory */
      break;
      case DSP_EXECUTE:
        /* Tell DSP to Execute – Assume code writes to result_buffer */

        mcapi_pktchan_send(data_send_chan, result_buffer, result_size,
                           &status);
        CHECK_STATUS(status);
      break;
      case DSP_TERMINATE:
        dsp_shutdown();
      break;
      default:
      break;
    }
    command = receive_dsp_cmd();
  }
}
```

7.5 Packet Processing

This example presents the typical startup and inner loop of a packet processing application. There are two source files: load_balancer.c and worker.c. The main entrypoint is in load_balancer.c.

The program begins in the load balancer, which spawns a set of worker processes and binds channels to them. Each worker has two channel connections to the load balancer: a packet channel for work requests and a scalar channel for acks. When work arrives on the packet channel from the load balancer, the worker processes it and then sends back an ack to the load balancer. The load balancer will not send new work to a worker unless an “ack” word is available.

This example uses both packet channels and scalar channels. Scalar channels are used for acks from the workers to the load balancer, since each ack is a single word and performance will be better without the packetization overhead. Packet channels are used to send work requests to the workers, since each work request is a structure containing multiple fields. The example also shows how to use both statically named and dynamic (anonymous) endpoints; it uses the endpoint creation functions mcapi_create_endpoint(), mcapi_create_anonymous_endpoint(), and mcapi_get_endpoint().

On architectures with hardware support for scalars it would make more sense to use scalar channels for the work requests as well. Work requests are usually composed of only a few words, so they can be reasonably sent as individual words. More importantly, the load balancer is the rate-limiting step in this program; we can always add more workers but they're useless if the load balancer can't feed them. Since the load balancer is limited by communication overhead, hardware accelerated scalar channels could potentially improve overall performance if the overhead of a packet channel send is greater than the comparable scalar channel sends.

7.5.1 Packet Processing Code

7.5.1.1 common.h

```
// Header file containing a couple of declarations that are shared
// between the load balancer and the worker.
#ifndef __EXAMPLE_COMMON_H__
#define __EXAMPLE_COMMON_H__

enum {
  WORKER_REQUEST_PORT_ID,
  WORKER_ACK_PORT_ID
};

#define CHECK_STATUS(S) do {  \
  if (S != MCAPI_SUCCESS) {    \
    printf("Failed with MCAPI status %d (%s)\n", S, mcapi_strerror(S)); \
    exit(1); \
 }} while (0)

#endif
```

7.5.1.2 load_balancer.c

```
#include <mcapi.h>
#include <stdbool.h>


// We will have four worker processes
static const int NUM_WORKERS = 4;

// An array of packet channel send ports for sending work descriptors
// to each of the worker processes.
mcapi_pktchan_send_port_t work_requests_out[NUM_WORKERS];

// An array of scalar channel receive ports for getting acks back from
// the workers.
mcapi_sclchan_recv_port_t acks_in[NUM_WORKERS];


// function declarations
void create_and_init_workers(void);
void dispatch_packets(void);
void shutdown_lb(void);

// The entrypoint for this packet processing application.
int main(void)
{
  mcapi_init(0);
  create_and_init_workers();
  dispatch_packets();
  shutdown_lb();

  return 0;
}


void create_and_init_workers()
```

```
{
  for (int i = 0; i < NUM_WORKERS; i++)
  {
    mcapi_status_t* status;

    // Spawn a new thread; pass parameters so the new thread will execute
    // worker_spawn_function(bootstrap_endpoint)
    spawn_new_thread(&worker_spawn_function, i);
    int worker_node = i + 1;

    // Create a send endpoint to send packets to the worker; get the
    // worker's receive endpoint via mcapi_get_endpoint()
    mcapi_endpoint_t work_request_out_endpoint =
      mcapi_create_anonymous_endpoint(&status);
    CHECK_STATUS(status);
    mcapi_endpoint_t work_request_remote_endpoint =
      mcapi_get_endpoint(worker_node, WORKER_REQUEST_PORT_ID, &status);
    CHECK_STATUS(status);

    // Bind the channel and open our local send port.
    mcapi_connect_pktchan(work_request_out_endpoint,
                          work_request_remote_endpoint, &status);
    CHECK_STATUS(status);
    work_requests_out[i] =
      mcapi_open_pktchan_send(work_request_out_endpoint, &status);
    CHECK_STATUS(status);


    // Repeat the process to create an ack scalar channel from the
    // worker back to us.
    mcapi_endpoint_t ack_in_endpoint =
      mcapi_create_anonymous_endpoint(&status);
    CHECK_STATUS(status);
    endpoint_t ack_remote_endpoint =
      mcapi_get_endpoint(worker_node, WORKER_ACK_PORT_ID, &status);
    CHECK_STATUS(status);

    mcapi_connect_sclchan(ack_remote_endpoint, ack_in_endpoint, &status);
    CHECK_STATUS(status);

    acks_in[i] = mcapi_open_sclchan_recv(ack_in_endpoint, &status);
    CHECK_STATUS(status);
  }
}


void dispatch_packets()
{
  PacketInfo packet_info;
  mcapi_status_t* status;

  while (get_next_packet(&packet_info))
  {
    // Because we maintain "session state" across packets, each
    // incoming packet is associated with a particular worker.
    int worker = packet_info.worker;

    // Each worker sends back acks when it is ready for more work.
    if (mcapi_sclchan_available(acks_in[worker], &status))
    {
      // An ack is available; pull it off and send more work
```

```
      int unused_result = mcapi_sclchanrecv_uint8(acks_in[worker]);
      mcapi_pktchan_send(work_requests_out[worker], &packet_info,
                         sizeof(packet_info), &status);
      CHECK_STATUS(status);
    }
    else
    {
      // No ack available; drop the packet (or queue, or do some other
      // form of exception processing) and move on.
      drop_packet(&packet_info);
    }
  }
}


void shutdown_lb()
{
  mcapi_status_t status;

  // Shutdown each worker in turn
  for (int i = 0; i < NUM_WORKERS; i++)
  {
    // Send an “invalid” packet to trigger worker shutdown
    PacketInfo invalid_work;
    invalid_work.valid = false;
    mcapi_pktchan_send(work_requests_out[i], &invalid_work,
                       sizeof(invalid_work), &status);
    CHECK_STATUS(status);

    // Close our ports; don't worry about errors (what would we do?)
    mcapi_pktchan_send_close(work_requests_out[i], &status);
    mcapi_sclchan_receive_close(acks_in[i]);
  }
}
```

7.5.1.3 worker.c

```
#include <mcapi.h>
#include <stdbool.h>

// Local port for incoming work requests from the load balancer.  We
// use a packet channel because each work request is a structure with
// multiple fields specifying the work to be done.
mcapi_pktchan_recv_port_t work_request_in;

// Local port for outgoing acks to the load balancer.  We use a scalar
// channel because each ack is a single word indicating the number of
// work items that have been completed.
mcapi_sclchan_send_port_t ack_out;

// a local buffer for use by the incoming packet channel
char work_channel_buffer[1024];


// function declarations
void bind_channels(endpoint_t load_balancer_endpoint);
void do_work(void);
void shutdown(void);

// The function called within each new worker thread.  This function
```

```
// binds takes an endpoint in the load balancer as its parameter so
// that it can communicate with the load balancer and create channels
// between the worker and the load balancer.
void worker_spawn_function(int worker_num)
{
  mcapi_init(1 + worker_num);
  bind_channels();
  do_work();
  shutdown();
}

void bind_channels()
{
  mcapi_status_t status;

  // Create a packet receive endpoint for incoming work requests; use
  // a static port number so that load balancer can look it up via
  // mcapi_get_endpoint().
  mcapi_endpoint_t work_request_in_endpoint =
    mcapi_create_local_endpoint(WORKER_REQUEST_PORT_ID, &status);
  CHECK_STATUS(status);

  // The load balancer will bind a channel to that endpoint; we use
  // the open function to synchronize and initialize our local packet
  // receive port.
  mcapi_open_pktchan_recv(&work_request_in, work_request_in_endpoint,
                                            &status);
  CHECK_STATUS(status);

  // Repeat the process to create a scalar channel from us back to the
  // load balancer.
  mcapi_endpoint_t ack_out_endpoint =
    mcapi_create_local_endpoint(WORKER_ACK_PORT_ID, &status);
  CHECK_STATUS(status);

  ack_out = mcapi_open_sclchan_send(ack_out_endpoint, &status);
  CHECK_STATUS(status);
}


void do_work()
{
  mcapi_status_t status;

  // The first thing we do is send an ack so that the load balancer
  // knows we're ready for work.
  mcapi_sclchan_send_uint8(ack_out, 1 /* Any value would do. */, &status);

  while (1)
  {
    // receive a work request from the load balancer
    PacketInfo* work_info;
    mcapi_pktchan_recv(work_request_in, &work_info, NULL, &status);
    CHECK_STATUS(status);

    // if the work request is marked “invalid”, we're done
    if (work_info->is_valid == false)
    {
      break;
    }
```

```
    // do the work
    some_function_or_another(work_info);

    // we're done; return the buffer and send an ack
    mcapi_pktchan_free(work_info)
    mcapi_sclchan_send_uint8(ack_out, 1 /* Any value would do. */, &status);
  }
}


void shutdown()
{
  mcapi_status_t status;

  // close our channels
  mcapi_pktchan_recv_close(work_request_in, &status);
  mcapi_sclchan_send_close(ack_out, &status);
}
```

8 Acknowledgements

The MCAPI working group would like to acknowledge the significant contributions of the following people in the creation of this API specification:

Working Group
Ajay Kamalvanshi, Nokia Siemens Networks
Anant Agarwal, Tilera
Sven Brehmer, PolyCore Software
Max Domeika, Intel
Peter Flake, Imperas
Steven Furr, Freescale Semiconductor, Inc.
Masaki Gondo, eSOL
Patrick Griffin, Tilera
Jim Holt, Freescale Semiconductor, Inc.
Rob Jackson, Imperas
Kevin Kissell, MIPS
Maarten Koenig, Wind River
Markus Levy, Multicore Association
Charles Pilkington, ST Micro
Andrew Richards, Codeplay
Colin Riley, Codeplay
Frank Schirrmeister, Imperas

The MCAPI working group also would like to thank the external reviewers who provided input and helped us to improve the specification below is a partial list of the external reviewers (some preferred to not be mentioned).

Reviewers
David Addison, ST Micro
Sterling Augustine, Tensilica
Badrinath Dorairajan, Aricent
Marc Gauthier, Tensilica
David Heine, Tensilica
Dominic Herity, Redplain Technology
Arun Joseph, IBM
Michael Kardonik, Freescale Semiconductor, Inc.
Anjna Khanna, Cadence Design Systems
Grant Martin, Tensilica
Tim Mattson, Intel
Dror Maydan, Tensilica
Girish Mundada, Micronas
Karl Mörner, Enea
Emeka Nwafor, Zeligsoft
Don Padgett, Padgett Computing
Ola Redell, Enea
Michele Reese, Freescale Semiconductor, Inc.
Greg Regnier, Intel
Vedvyas Shanbhogue, Intel
Patrik Strömblad, Enea